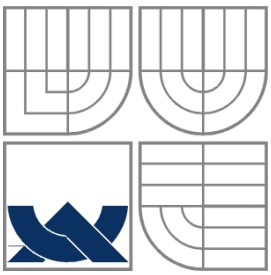
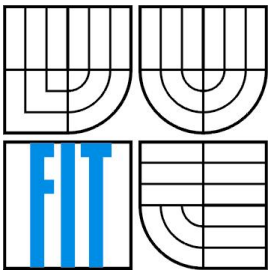


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

INTUITIVNÍ KRESLENÍ NA PLATFORMĚ ANDROID

INTUITIVE DRAWING ON THE ANDROID PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN APPL

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. ADAM HEROUT, Ph. D.

BRNO 2012

Intuitivní kreslení na platformě Android

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doc. Ing. Adama Herouta Ph. D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Bc. Martin Appl
21. května 2012

Poděkování

Děkuji doc. Ing. Adamu Heroutovi za rady a odbornou pomoc. Dále Štěpánu Křížkovi za zpětnou vazbu a pomoc se zpracováním grafiky.

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací programu ke kreslení prstem na mobilních zařízeních se systémem Android. Hlavní důraz byl kladen na promyšlené, intuitivní a přívětivé uživatelské rozhraní. Dále je řešena interpolace bodů křivkou, posun a změna velikosti plátna pomocí transformačních matic, rozsáhlá historie akcí a několik základních nástrojů.

Abstract

This master's thesis deals with design and implementation of finger painting application for mobile devices with Android operating system. Main focus is on well designed, intuitive and friendly user interface. Solved problems are spline interpolation of points, zoom and pinch with transformation matrices, extensive history for action reversal and few basic tools.

Klíčová slova

Android, Kreslení, Uživatelské rozhraní, 2D grafika, Interpolace, Spline křivky, Štětce, Transformační matice

Keywords

Android, Drawing, User interface, 2D graphics, Interpolation, Splines, Brushes, Transformation matrices

Citace

Martin Appl: Intuitivní kreslení na platformě Android, diplomová práce, Brno, FIT VUT v Brně, 2012

Obsah

1 Úvod

Dnešní kapesní elektronická zařízení, jako jsou telefony a tablety, poskytují umělcům zajímavou možnost, jak mít neustále po ruce silný nástroj na zachycení svých myšlenek a inspirace.

Nelze předpokládat, že by se grafická aplikace na mobilním telefonu mohla vyrovnat grafické pracovní stanici s tabletem. Zaměření aplikace by tedy mělo být spíše na rychlost práce a intuitivnost než na množství funkcí. Uživatel pak může zachycené myšlenky dále upravit, zvláště pokud mu jsou poskytnuty dobré prostředky pro export.

Tato diplomová práce se zabývá návrhem a implementací aplikace umožňující kreslení prstem pro mobilní zařízení se systémem Android. Velký důraz je kladen především na promyšlené a intuitivní uživatelské rozhraní. Usnadnění často prováděných akcí kreslíře by mělo být spolu s příjemným vzhledem a interaktivitou hlavní přidanou hodnotou této aplikace. Nejprve však bylo nutné implementovat samotné základní funkce programu, což zabralo většinu objemu provedené práce.

Na začátku návrhu budoucí aplikace jsem otestoval a vyhodnotil několik konkurenčních aplikací dostupných v té době na Android Marketu. Snažil jsem se zjistit například typická paradigmata ovládání, na která už jsou uživatelé zvyklí a bylo by tedy vhodné je převzít. Dále jsem sledoval různé neduhy, abych se jich mohl vyvarovat. Shrnutí těchto poznatků nabízí kapitola 2.

Kapitola 3 provede čtenáře teorií nutnou k pochopení algoritmů interpolace bodů a vykreslování stopy štětce, včetně jejich návrhu a obecného popisu. Dále se zabývá afinními transformacemi a jejich maticovou reprezentací, což jsou výchozí znalosti pro implementaci změny velikosti a posunutí plátna.

Kapitola 4 nabízí stručnou charakteristiku platformy Android a jejího SDK. Vybrány jsou jednak základní stavební bloky společné většině aplikací, jednak různé třídy a rozhraní, které se budou hodit specificky pro návrh této aplikace. Čtenář by měl získat základní představu o architektuře aplikací a o způsobu vývoje nových a úpravě stávajících komponent.

Kapitola 5 se zabývá implementací a konkrétním využitím algoritmů z kapitoly 3. Je tu popsána architektura jednotlivých modulů aplikace. Čtenář se zde dozví, které třídy zajišťují jaké funkce aplikace a princip, jak pracují.

Nakonec zbývá těžiště celé práce, jímž je uživatelské rozhraní. Jeho návrh a implementace je obsahem kapitoly 6. Rozsah textu v tomto dokumentu příliš neodpovídá rozsahu vykonané práce. Je to z toho důvodu, že se většinou jedná o mírné úpravy stávajících komponent a jemné ladění různých vlastností, které nejsou pro čtenáře tak zajímavé. Snažil jsem se tedy soustředit spíše na obecně platné zásady a cíle při návrhu. Popsány jsou zároveň funkce a principy implementace prvků, které nebyly v rámci platformy dostupné a musel jsem je k dosažení stanovených cílů vytvořit. Pro detailní pochopení bych čtenáře odkázal na zdrojové kódy a dokumentaci na přiloženém paměťovém médiu.

2 Možnosti intuitivního kreslení na platformě Android

Tato kapitola se zabývá průzkumem vlastností konkurenčních aplikací dostupných na platformě Android. Výsledky budou sloužit především k větší konzistenci návrhu ovládání s existujícími řešeními. Dále je určitě vhodné se inspirovat případnými dobrými vlastnosti a vyvarovat se stejných chyb. Většina zde uvedených aplikací bude hodnocena podle volně dostupných demoverzí. U produktů, kde jsem měl k dispozici plnou verzi to explicitně uvedu.

2.1 Andraw

Program nabízí zajímavou funkci volby pozice kurzoru. Je možné si zvolit umístění buď pod středem bříška prstu nebo kousek nad nehtem, kde už nebude prst zakrývat obrazovku.

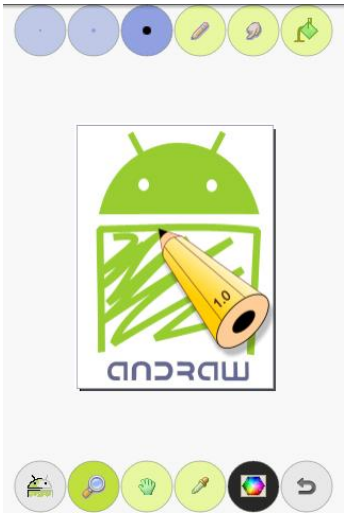
Umístění kurzoru nad nehtem se zdá být výhodné pro přesnost, ale na druhou stranu je při započetí čáry těžké uhodnout, kde se kurzor objeví, což může být při rychlé práci nepříjemné. Pokud potřebujeme pracovat na jemných detailech, je dle mého názoru lepší využít zoomu, proto bych označil volbu kurzoru pod bříškem prstu jako intuitivnější pro uživatele.

Zoom je v tomto programu ale velmi špatně vyřešen, takže volba pozice kurzoru má zde jistě smysl. Po aktivaci tlačítka uživatel táhne jedním prstem a program kolem počátečního bodu gesta přiblíží obraz na předem stanovenou hodnotu. Chování této funkce je poměrně nevypočitatelné. Pravděpodobně bylo navrženo s ohledem na zařízení bez funkce multitouch.

Celkově aplikace nabízí jeden štětec ve třech přednastavených velikostech s několika nastaveními stopy, nástroj rozmazání, nástroj plechovka barvy, nástroj kapátko, dále přiblížení, posunutí plátna a historii o několika krocích. Nejzajímavější funkcí je přehrávání postupu kreslení ve formě animace.

Při převodu pozice bodů na obrazovce na dráhu čáry chybí interpolace, takže při rychlém tahu kdy systém nedodává body dostatečně rychle vznikají na čáře ostré hrany.

Grafické uživatelské rozhraní je vyřešeno pěkně. Jen tlačítka ubírají místo z plochy plátna.



Obrázek 1:Uživatelské rozhraní Andraw

Klady	Zápory
Intuitivní GUI	Chybí interpolace
Dobře fungující animace průběhu kresby	Málo možností
	Špatný zoom

Tabulka 1: Klady a zápory aplikace Andraw

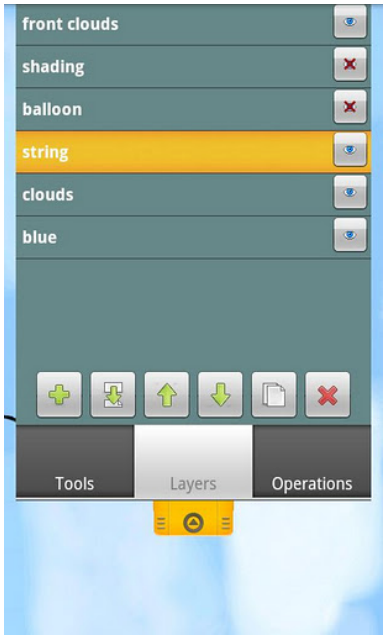
2.2 Canvas

Na první pohled působí tato aplikace velmi propracovaným dojmem. Menu je vyřešeno pomocí panelu nástrojů, který zobrazíme tažením za držadlo, umístěné při horním okraji obrazovky. Nabídku je možné přepínat mezi záložkami nástrojů, vrstev a operací s hotovým obrázkem. Celkově je ovládání poměrně intuitivní. Využívá však některých prvků GUI knihovny poněkud nestandardně a význam mnoha tlačítek není na první pohled zřejmý. Rozhraní by mohlo vypadat vizuálně lépe. Působí poslepovaným dojmem.

Funkce přiblížení a posouvání plátna se ovládají pomocí gesta dvěma prsty, což je u kvalitních aplikací standard.

Bohužel opět chybí interpolace, takže i tato aplikace tvoří zubaté čáry. Výsledek kreslení štětcem sám o sobě nevypadá příliš dobře a hrany vznikající při rychlém tahu rozhodně na kráse nepřidávají.

Aplikace je zaměřená spíše na práci s vrstvami a jako jedna z mála podporuje označování a kopírovaní částí obrazu. Počítá se tedy zřejmě s jejím použitím spíše pro editaci než pro tvorbu.



Obrázek 2: Uživatelské rozhraní Canvas s aktivní záložkou vrstev

Klady	Zápory
Práce s vrstvami	Chybí interpolace
Nástroj laso	Menu místy nepřehledné

Tabulka 2: Klady a zápory aplikace Canvas

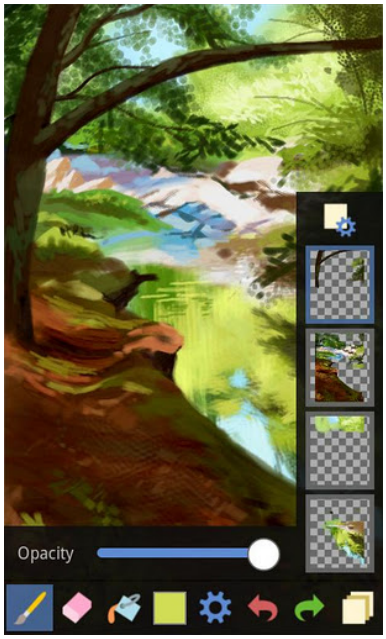
2.3 Fresco

Na rozdíl od předešlých programů má Fresco promyšlené uživatelské rozhraní. Po spuštění uživatele uvítá obrazovka galerie, kde jsou náhledy předešlých prací. Je možné je otevřít stisknutím.

Fresco má také alespoň nějakou interpolaci a čáry nejsou tak hrubé jako u některých konkurentů. Nicméně řešení není úplné a pokud provedeme dostatečně rychlý tah, můžeme stále vidět body propojené přímkami.

Naopak dobře propracované jsou tu vrstvy. Práce s nimi je jednoduchá a intuitivní.

Z dalších funkcí uvádím například filtry obrazu, různé transformace jako zrcadleni nebo otočení obrazu, úpravu barev a v neposlední řadě export zejména do Adobe® Photoshop® formátu (.PSD) , který je dostupný v placené verzi.



Obrázek 3: Uživatelské rozhraní Fresco

Klady	Zápory
Rozhraní pro práci s vrstvami	Nedokonalá interpolace bodů
Jednoduché a hezké GUI	Nepřesné posouvání plátna
Možnosti exportu	Žádná paměť barev
Korekce barev, transformace a filtry	
Galerie děl pro rychlé otevření	

Tabulka 3: Klady a zápory aplikace Fresco

2.4 Magic Doodle

Magic Doodle se vymyká především velkým množstvím funkcí. Autorům nelze upřít spoustu dobrých a užitečných nápadů. Implementace některých vlastností však obsahuje chyby. Na uživatelském rozhraní je poznat, že nebylo promyšleno dopředu, ale postupně se nabalovalo na sebe, jak přibývaly funkce. Chybí mu sjednocující prvek, takže některá podmenu se chovají a vypadají jinak.

Zajímavé jsou procedurální nástroje, které generují podél dráhy kresby různé efekty s využitím náhodnosti nebo rychlosti tahu.

Práce s vrstvami nabízí vše, co je potřeba. K dispozici je velké množství štětců a dalších nástrojů. U štětců je vizuální kvalita trochu horší než u konkurence. Aplikace sdružuje téměř všechny funkce konkurenčních aplikací na jednom místě. S propracovanějším GUI a lepší kvalitou štětců by byla asi nejlepší, kterou jsem nalezl.

Obrázek 4: Uživatelské rozhraní Magic Doodle

Klady	Zápory
Velké množství funkcí	Nepřehledné GUI
Přehrávání průběhu kresby	Chyby v zoomu
Funkce zrcadlení	Vhled stop štětců
Dostupné naposledy použité nástroje a barvy	

Tabulka 4: Klady a zápory Magic Doodle

2.5 SketchBook® Mobile

SketchBook® Mobile je profesionální aplikace od společnosti Autodesk. Z testovaných aplikací udělala nejlepší dojem. Ve volně dostupné verzi je omezena jen na některé nástroje, ale plná verze nabízí ve vysoké kvalitě vše, co lze najít u konkurence. Dostupná je rovněž verze optimalizovaná speciálně pro tablety, tu jsem ale neměl k dispozici.

Program nabízí mimo rozsáhlé palety běžných nástrojů a funkcí i galerii prací s možností importu a exportu, včetně Photoshop® formátu.

Obrázek 5: Uživatelské rozhraní SketchBook® Mobile

Klady	Zápory
Vhled uživatelského rozhraní	Časté operace zaberou hodně kliknutí
Intuitivnost ovládání	Nemá animaci průběhu kresby
Kvalita štětců a nástrojů	Absence historie barev a nástrojů
Široké možnosti nastavení	

Tabulka 5: Klady a zápory SketchBook® Mobile

2.6 Shrnutí běžně dostupných funkcí kreslících programů

Všechny testované programy mají základní funkce podobné, liší se však v kvalitě provedení a v kvalitě návrhu uživatelského rozhraní.

Obvykle aplikace obsahuje několik různých štětců. Jednoduché aplikace obsahují pouze několik základních nástrojů. Pokročilé umožňují širokým nastavením docílit v podstatě neomezeného počtu různých druhů efektů.

Typickými nástroji, které najdeme téměř u všech testovaných programů, jsou guma, plechovka barvy, kapátko, lupa a posunutí plátna.

Lepší programy nabízí navíc práci s vrstvami, zrcadlení při kreslení, podporu geometrických tvarů a pokročilejší nástroje, například rozmazání nebo různé efekty a filtry.

2.7 Přehled typických prvků v ovládání

Zoom je velmi často ovládán pomocí gest dvěma prsty, jak je běžné na zařízeních s dotykovým displejem se schopností sledovat více dotyků. U kreslících aplikací je posunutí obrazu, které se běžně

provádí jedním prstem, změněno na gesto dvěma prsty, kdy prsty udržují téměř konstantní vzdálenost od sebe. Tažení jedním prstem se totiž využívá při kreslení čar.

Některé programy využívají k ovládaní i dalších gest. Podržení prstu na místě například aktivuje nástroj kapátko. Dříve bylo ovládání tímto gestem i v aplikaci SketchBook®. V dalších verzích bylo odstraněno. Domnívám se, že může docházet k nechtěným aktivacím.

SketchBook® využívá dvojkliku v rozích obrazovky k ovládaní čtyř často používaných funkcí. Myslím, že je to dobrý nápad a navíc dvojkliku na plátně by se dalo také využít.

Nastavení typu štětce společně s parametry bývá často řešeno modálním dialogem, který znemožní přístup k plátnu, dokud ho nezavřeme. Některé aplikace nabízí zkratku na často používané nástroje včetně štětců. Dle mého názoru to hodně pomáhá zjednodušit a zrychlit práci s uživatelským rozhraním.

Volba barev vypadá podobně. Většinou však nejsou žádné rychle dostupné předvolené. Použití odstínů u uživatelů je velmi různorodé, takže vybrat část barev asi nemá smysl. Většinou se používá rozsáhlejší paleta. Smysl by podle mě mohly mít rychle dostupné naposledy použité barvy. Uživatelé se totiž někdy potřebují vrátit k dříve používanému odstínu a s kapátkem je to zdlouhavé.

Zejména u tabletů, kde je dostupná velká plocha obrazovky a dobré rozlišení, by se hodila možnost nechat menu zobrazená při práci. Zrychlilo by to přístup k funkcím. Je třeba pamatovat, že plátno by stále mělo zabírat největší část pracovní plochy. Pokud ovládání překrývá větší část plátna, je výhoda rychlejšího přístupu vynulována menší plochou pro vlastní práci. To je problém u testovaných aplikací, které využívají toto řešení. Ideální by podle mě bylo zkombinovat velké modální menu s menšími stále zobrazenými panely rychlé volby.

Některé z testovaných aplikací umožňují i práci s vrstvami. Ovládání se obvykle skládá z miniatur obsahu jednotlivých vrstev seřazených podle toho, jak se překrývají, a z tlačítek k ovládání akcí, které s nimi lze provádět. Jedná se zejména o změnu pořadí, odstranění vrstvy, sloučení dvou vrstev do jedné, vytvoření nové a kopírovaní existující vrstvy. Často bývá také přítomen jezdec, kterým lze upravit průhlednost. Některé programy podporují i různé módy prolínání barev.

3 2D rastrová grafika a interpolace bodů křivkou

Aplikace, které se snaží provádět interpolaci nebo alespoň zvětšit počet bodů, podle kterých vykreslují čáry, dosahují vizuálně lepších výsledků.

Tato kapitola se zabývá teorií nutnou pro pochopení a návrh algoritmů interpolátoru, jakožto i dalších modulů, souvisejících s vykreslováním grafiky na plátno.

Dále je obsažena teorie afinních transformací a jejich maticová reprezentace, což jsou důležité výchozí znalosti pro implementaci funkcí pro posun a změnu velikosti zobrazení pracovní bitmapy.

3.1 Bézierovy křivky

Bézierova křivka je parametrická křivka definovaná body (P_0,P_1, … , P_n), kde body P_0 a P_n jsou koncové body. Ostatní body se nazývají řídící a udávají tvar křivky. Zakřivení se chová, jako kdyby bylo přitahováno k řídícím bodům. U racionálních Bézierových křivek je navíc každému řídícímu bodu přiřazeno reálné číslo, které určuje do jaké míry daný bod ovlivní výsledný tvar.

Bézierova křivka stupně *n* je definována *n+1* kontrolními body a její obecný tvar můžeme zapsat vzorcem (3.1).

$$B(t)=\sum_{i=0}^{n}\binom{n}{i}(1-t)^{n-i}t^{i}P_{i} \qquad t\in\langle 0,1\rangle \tag{3.1}$$

Kde (3.2) je *i-tý* Bernsteinův polynom *n-tého* stupně.

$$\binom{n}{i}(1-t)^{n-i}t^{i} \qquad i=0,1,\ldots n \tag{3.2}$$

Dosazením různých hodnot parametru *t* do rovnice zjistíme souřadnice bodů ležících na křivce, přičemž *t = 0* odpovídá počátečnímu bodu a *t = 1* koncovému.

Pokud budeme dosazovat za *t* různé hodnoty s konstantním přírůstkem, nedostaneme body ležící ve stejné vzdálenosti od sebe. Vzdálenost bodů je závislá na zakřivení křivky.

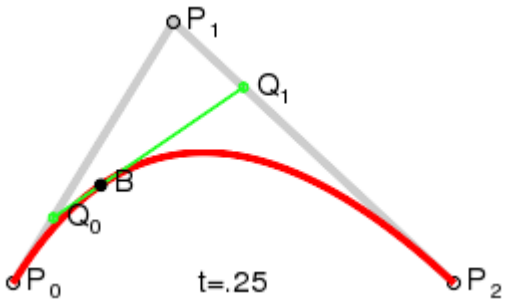
K rasterizaci Bézierových křivek se v počítačové grafice používá rekurzivní algoritmus de Casteljau. Netrpí totiž výše uvedenou nevýhodou.

3.1.1 Kvadratické Bézierovy křivky

Jsou křivky druhého stupně (*n=2*). Dráha je popsána funkcí (3.3).

$$B(t)=\sum_{i=0}^{2}\binom{2}{i}(1-t)^{2-i}t^{i}P_{i}=(1-t)^{2}P_{0}+2t(1-t)P_{1}+t^{2}P_{2},\ t\in\langle 0,1\rangle \tag{3.3}$$

Příklad konstrukce kvadratické Bézierovy křivky dělením úseček řídícího polynomu dle algoritmu de Casteljau vidíme na obrázku 6.



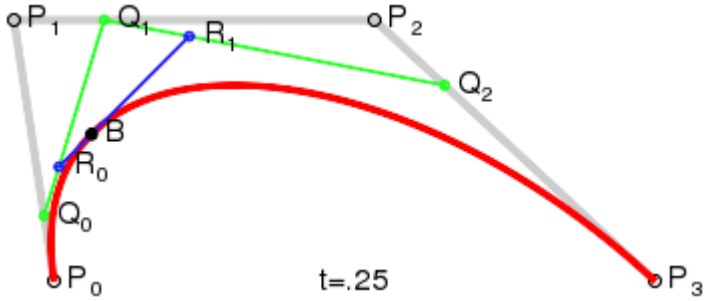
Obrázek 6: Konstrukce kvadratické Bézierovy křivky

3.1.2 Kubické Bézierovy křivky

Tímto pojmem rozumíme křivky třetího stupně (*n=3*). V praxi jsou velmi často používané, jelikož je lze snadno spojovat do spline křivek. Popsat je můžeme funkcí (3.4).

$$B(t)=\sum_{i=0}^{3}\binom{3}{i}(1-t)^{3-i}t^{i}P_i=(1-t)^3P_0+3t(1-t)^2P_1+3t^2(1-t)P_2+t^3P_3 \qquad (3.4)$$
$$t\in\langle 0,1\rangle$$

Příklad konstrukce vidíme na obrázku 7.



Obrázek 7: Konstrukce kubické Bézierovy křivky

Body označené Q_i se pohybují po lineárních Beziérových křivkách, což jsou v podstatě úsečky. Body R_i udávají svojí dráhou kvadratické křivky a dráha bodu B vykresluje výslednou kubickou křivku. Více viz. [1].

3.2 Spline křivky

Pojmem spline označujeme v matematickém oboru numerické analýzy speciální polynom rozdělený na části. Interpolace pomocí spline je preferována před polynomiální interpolací, protože nedochází k takzvanému Rungeho fenoménu, což je zvlnění na krajích interpolované číselné řady.

Název spline pochází z pružného pravítka, které před objevem parametrických křivek používali inženýři ke kreslení oblých tvarů procházející sadou definovaných bodů.

Spline křivky můžeme rozdělit na interpolační a aproximační. Interpolační křivky prochází danou sadou bodů, přičemž body leží přímo na křivce. Aproximační se používají při návrhu. Zadané doby určují tvar křivky, ale neleží na ní. Řídící polygon je vlastně hrubou aproximací tvaru výsledné křivky. Tu pak můžeme dále zjemňovat, což je například princip algoritmu de Castejau.

3.2.1 Obecný B-spline

B-spline je zkratka pro „*basis spline*“. V podstatě se jedná o zobecnění Bézierových křivek. Máme-li *m* reálných čísel t_i zvaných uzly pro které platí (3.5), pak B-spline stupně *n* je parametrická po částech polynomiální křivka složená lineární kombinací základních křivek $b_{i,n}$ stupně *n*.

$$t_0 \leq t_1 \leq \ldots t_{m-1} \tag{3.5}$$

$$S(t) = \sum_{i=0}^{m-n-2} P_i b_{i,n}(t),\ t \in \langle t_n, t_{m-n-1} \rangle \tag{3.6}$$

Body P_i nazýváme kontrolní. Máme *m-n-1* kontrolních bodů a společně tvoří konvexní obálku křivky.

Pokud je vzdálenost mezi jednotlivými uzly konstantní, čili platí (3.7), říkáme, že je spline uniformní. Pro neuniformní spline vyplývá jediné omezení pro t_i z (3.5).

$$t_{i+1} - t_i = const \tag{3.7}$$

Můžeme narazit ještě na dělení křivek na racionální a neracionální. Přičemž neracionální křivky jsou vlastně speciálním případem racionálních. Racionální křivky mají u kontrolního bodu přiřazenou váhu, která ovlivňuje jeho míru vlivu na výsledný tvar. Neracionální jsou tedy případ, kdy je váha u všech bodů stejná.

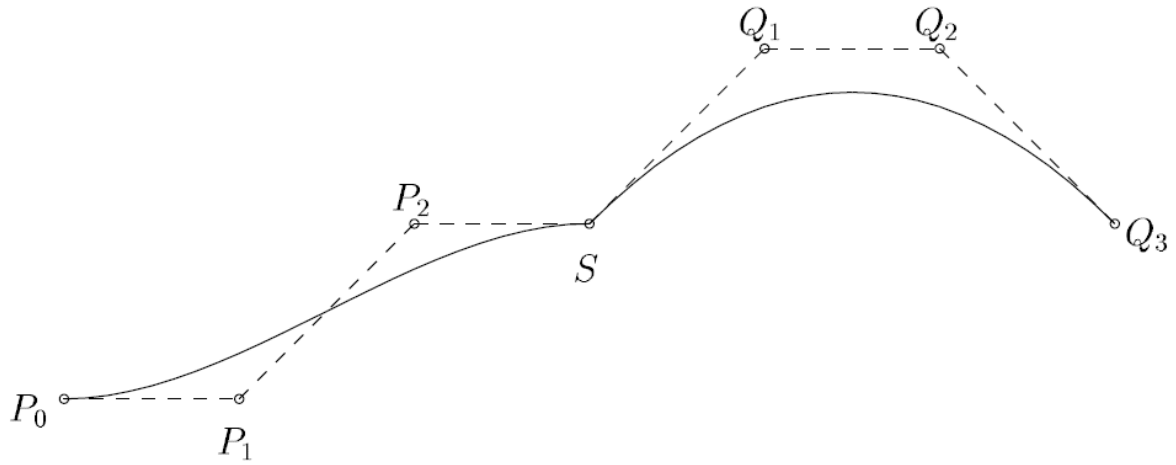
V počítačové grafice se nejčastěji používá neuniformní racionální B-spline křivka (NURBS).

Uniformní B-spline stupně *n* je posloupnost Bézierových křivek stupně *n* napojených v koncových bodech se spojitostí C^{n-1}.

Vzhledem k podmínkám spojitosti, kterým se budeme věnovat v kapitole 3.2.2, potřebujeme k definování splinu složeného z *L* křivek stupně *n* právě *L* + *n* -1 bodů [2].

3.2.2 Spojování Bézierových křivek

Pokud chceme vytvořit spojitý spline z několika Bézierových křivek, musíme nejprve zajistit napojení v koncových bodech. Mějme tedy první křivku s kontrolními body P_0, P_1, P_2, P_3 a druhou s body Q_0, Q_1, Q_2, Q_3. Bod, ve kterém křivky spojíme, budeme pro přehlednost nazývat *S* (*S* = P_3 = Q_0). Tato spojitost se označuje C^0. Pokud bychom se dívali na křivku jako na dráhu bodu v závislosti na čase, může v bodě *S* dojít ke skokové změně vektoru rychlosti i zrychlení.



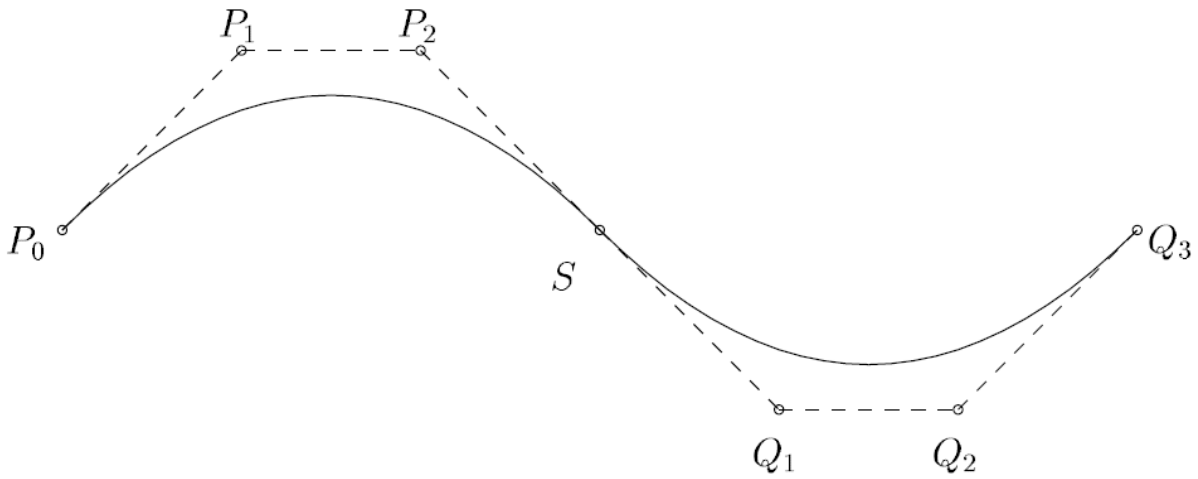
Obrázek 8: Spojení Bézierových křivek v koncovém bodě

Jak vidíme na obrázku 8, spojení nevypadá příliš přirozeně. Aby v místě napojení nebyla patrná hrana, musí se v bodě *S* shodovat první derivace u obou křivek.

První derivace v bodě *S* je u první křivky $3(S - P_2)$ u druhé $3(Q_1 - S)$. Z obrázku 8 lze rozeznat, že se vektory $S - P_2$ a $Q_1 - S$ neshodují ([1] vzorec 5.17) .

Abychom zajistili spojitost v první derivaci, musí být splněna podmínka (3.8). Pak bude platit, že se bod *S* nachází na středu úsečky P_2Q_1. Na obrázku 9 vidíme příklad.

$$S - P_2 = Q_1 - S \tag{3.8}$$



Obrázek 9: Spojení křivek spojité v první derivaci

Pokud bychom chtěli ještě pozvolnější napojení, zajistíme to shodou i ve druhé derivaci. V bodě *S* je parametr $t = 1$ pro první křivku a $t = 0$ pro druhou. Dosadíme-li do rovnice pro druhou derivaci kubické Bézierovy křivky (3.9) tyto hodnoty, dostaneme v bodě *S* dvě strany rovnice (3.10).

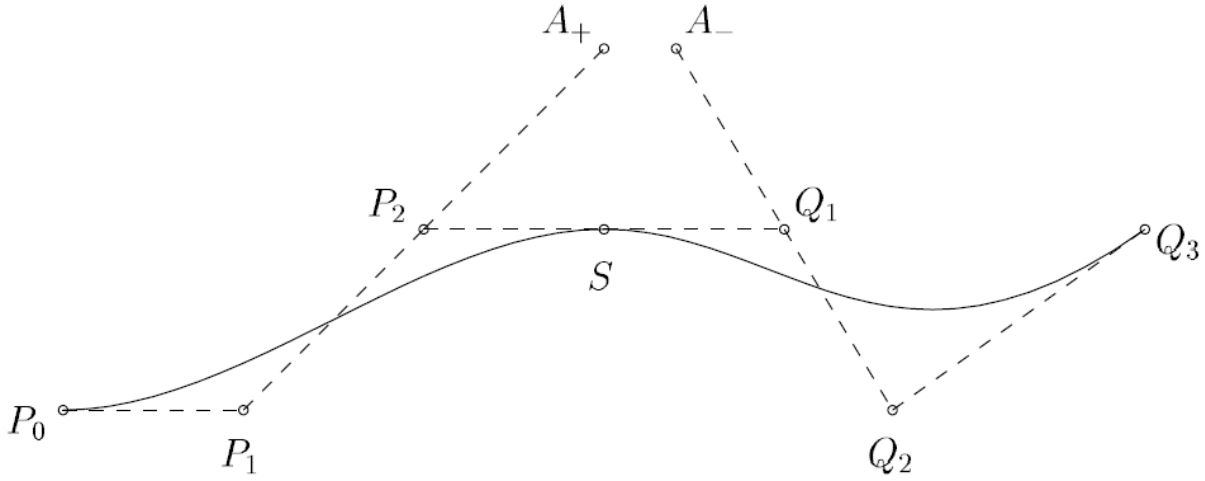
$$B''(t) = 6(1-t)P_0 + 3(6t-4)P_1 + 3(2-6t)P_2 + 6tP_3 \tag{3.9}$$

$$\begin{aligned} 6(P_1 - 2P_2 + S) &= 6(S - 2Q_1 + Q_2) \\ P_1 - 2P_2 &= Q_2 - 2Q_1 \end{aligned} \tag{3.10}$$

Znegujeme-li obě strany (3.10), můžeme se na každou ze stran rovnice dívat jako na bod. Pojmenujeme je A_+ a A_-, přičemž A_+ je pravý vrchol prvního řídícího polygonu, podobně A_- je levý vrchol druhého řídícího polygonu.

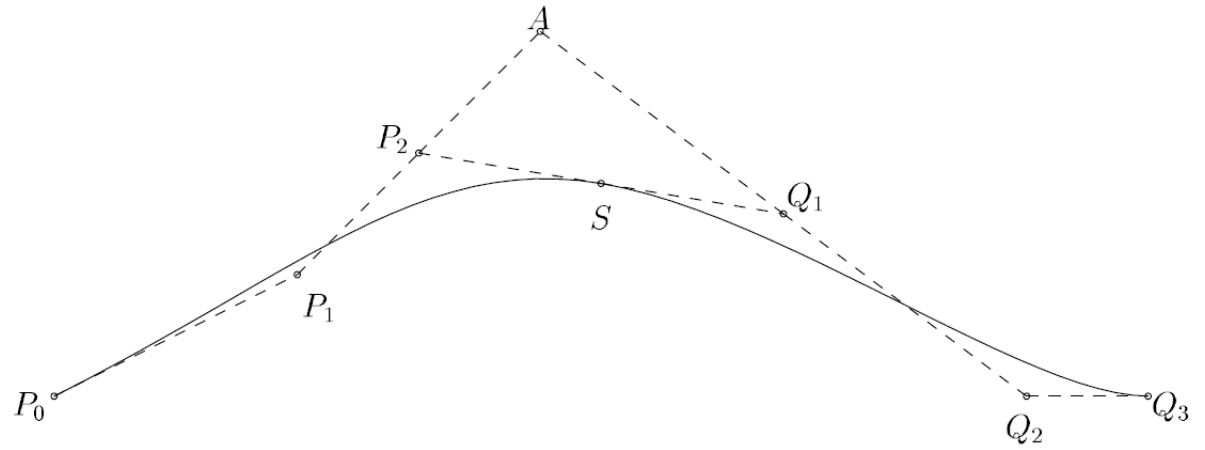
$$\begin{aligned} A_+ &= 2P_2 - P_1 = P_2 + (P_2 - P_1) \\ A_- &= 2Q_1 - Q_2 = Q_1 + (Q_1 - Q_2) \end{aligned} \tag{3.11}$$

Na obrázku 10 vidíme případ, kdy rovnice není splněna. Body A_+ a A_- se tedy nerovnají.



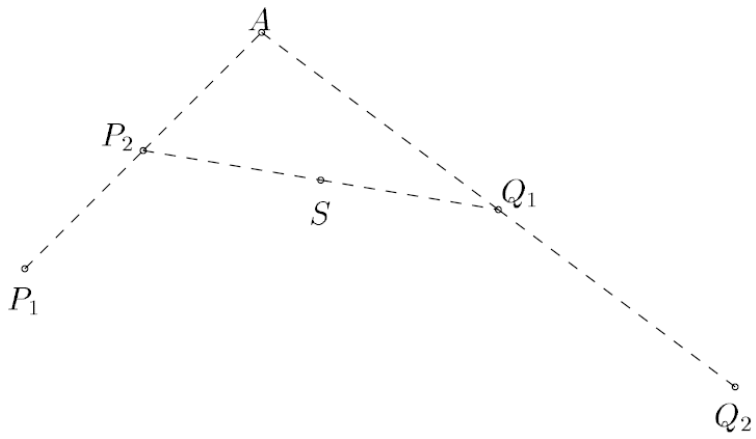
Obrázek 10: Vrcholy řídícího polygonu

Na obrázku 11 je pak případ, kdy je rovnice splněna a oba vrcholy řídícího polygonu jsou v jednom bodě A.



Obrázek 11: Spojení křivek spojité v druhé derivaci

Vidíme, že bod S je středem úsečky P_2Q_1, P_2 je středem P_1A a stejně tak Q_1 je středem Q_2A. Tyto body tvoří strukturu připomínající písmeno A, která se nazývá *A-frame*.



Obrázek 12: A-frame

Bézierovy křivky spojené v bodě *S* jsou spojité v první a druhé derivaci pouze pokud jejich kontrolní polygony tvoří *A-frame*.

3.2.3 Uvolněný uniformní kubický B-spline

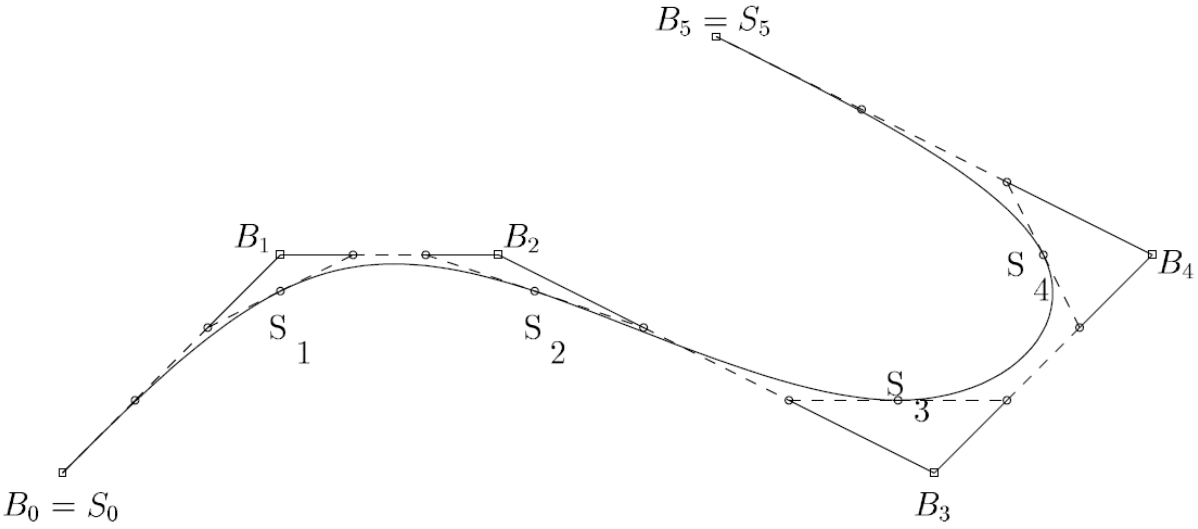
B-spline je definován body kontrolního polygonu B_0, B_1, … B_n.

Slůvko uvolněný (*relaxed*) v názvu znamená, že derivace v koncových bodech spline křivky jsou rovny nule. $B''_0(0) = 0$ a $B''_n(1) = 0$.

Chceme-li dosáhnout křivky spojité v druhé derivaci, musíme najít kontrolní body Bézierových křivek tak, aby splňovali podmínky *A-frame*. Toho dosáhneme, když každý segment kontrolního polygonu, to znamená každou úsečku B_iB_{i+1}, rozdělíme na tři stejné díly a označíme dělící body. V okolí každého bodu B_i kromě prvního a posledního spojíme dva z každé strany nejbližší dělící body úsečkou a její střed označíme S_i. Tímto vznikne *A-frame* s vrcholem v bodě B_i [3].

Každá z Bézierových křivek bude mít kontrolní bod S_i, dva dělící body úsečky B_iB_{i+1} a nakonec kontrolní bod S_{i+1}. Na začátku a konci křivky platí $S_0 = B_0$, $S_n = B_n$. Příklad konstrukce uvedeným způsobem vidíme na obrázku 13. Lze na něm pěkně rozeznat, kde se nachází *A-framy*.

Důležitou vlastností B-spline křivek je lokální vliv kontrolních bodů. Pro kubický spline je každý bod na křivce ovlivněn maximálně čtyřmi nejbližšími kontrolními body. Z této vlastností vyplývá, že pro vykreslení části křivky není nutné znát její zbytek, což budeme potřebovat při jednoprůchodové interpolaci.



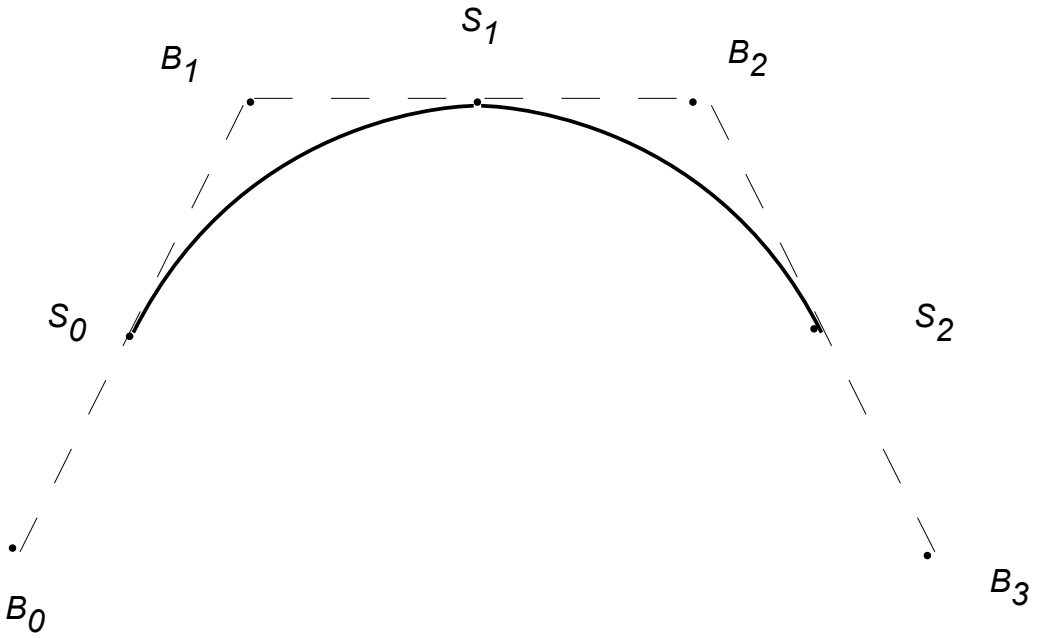
Obrázek 13: Konstrukce uvolněného uniformního kubického B-Spline

3.2.4 Uniformní kvadratický B-spline

Ačkoli se v praxi nepoužívá, můžeme podobně, jako jsme v kapitole 3.2.3 sestrojili kubický spline, sestrojit i kvadratický.

Vezmeme kontrolní body B_0, B_1, … B_n a do středu každé úsečky B_iB_{i+1} umístíme bod S_i, jak vidíme na obrázku 14. Pravidla spojitosti pro první derivaci jsou podobná, jako jsme viděli v kapitole 3.2.2 pro kubické křivky. Spojení koncových bodů se musí nacházet na středu přímky tvořené řídícími body navazovaných křivek. Toto tvrzení můžeme zapsat jako 3.12. Každá kvadratická Bézierova křivka bude mít řídící body S_{i-1},B_i, S_i.

$$S_i - B_i = B_{i+1} - S_i,\ i \in \langle 0, n \rangle \tag{3.12}$$



Obrázek 14: Konstrukce uniformního kvadratického B-spline

3.2.5 Spline vhodný pro interpolaci kurzoru při kreslení

V kreslící aplikaci obvykle dostaneme od API operačního systému proud bodů udávajících dráhu kurzoru. Aby mohl uživatel pohodlně pracovat, musíme zobrazovat výsledek okamžitě, jak dostaneme data. To vyřazuje z nabízených možností interpolační spline křivky, jejichž matematický aparát potřebuje vzít v úvahu uvolněné koncové podmínky, tedy rovnice pro druhé derivace v koncových bodech splinu. Jelikož potřebujeme vykreslovat v době, kdy ještě neznáme konec, je bohužel použití interpolačních křivek, které by kopírovaly vstupní body přesně, nemožné a musíme se spokojit s aproximačními, které budou v oblých částech od těchto bodů mírně uhýbat.

Při kreslení prstem můžeme předpokládat, že nebude často docházet k prudkým změnám směru. Velké množství segmentů křivky bude téměř přímkou. Toho můžeme využít k optimalizaci výkonu algoritmu interpolátoru.

Pro řešení je vhodný jak kvadratický, tak kubický uniformní B-spline. Kubický má samozřejmě spojitost i v druhé derivaci a kvadratický pouze v první. Vzhledem k předpokládanému typu vstupních dat bude ale pro uživatele málokdy patrný rozdíl. Kubický spline se bude držet blíže bodům řídícího polygonu. V implementační náročnosti téměř není rozdíl a s ohledem na tuto skutečnost bude v aplikaci využit kubický spline.

3.3 Princip štětců ve 2D kreslení

Základním principem algoritmu je otiskování hrotu štětce, což je malá bitmapa, podél dráhy nástroje v konstantní vzdáleností od sebe.

Konstantní vzdálenost bodů pro hrot štětce lze nejlépe udržet při renderování dráhy nástroje podél úseček navazujících koncovými body na sebe. Pro kreslení však potřebujeme umět renderovat i podél křivky. Algoritmus pracující přímo s matematickým popisem křivky by byl pravděpodobně značně netriviální. Nabízí se ale možnost nejprve rozdělit křivku na přiměřené množství úseček

a podél nich pak renderovat dráhu štětce. Pokud bude úseček dostatečný počet, nebude se výsledek téměř lišit od křivky. Pokud navíc bude délka úseček stejná nebo menší než vzdálenost hrotů štětce od sebe, aproximace nemůže dát horší výsledek, než kdybychom renderovali přímo podél křivky. Ačkoli může dojít k mírnému posunutí v závislosti na pozici otiskovaného bodu na přímce, na výsledném obrázku to nebude patrné.

3.3.1 Algoritmus vykreslující tah štětcem podél navazujících úseček

Nejprve se určí směrový vektor. Body A a B představují konce vykreslované úsečky. Výpočet se provádí pro každou ze souřadnic bodu zvlášť.

$$u_i = B_i - A_i \tag{3.13}$$

Vektor *u* dále normalizujeme na jednotkový vektor.

$$\hat{u} = \frac{u}{\|u\|} \qquad \|u\| = \sqrt{u_x^2 + u_y^2} \tag{3.14}$$

Písmenem *o* označíme zbytek vzdálenosti, kterou jsme neprošli u předchozí přímky v minulé iteraci. Celková vzdálenost k vykreslení bude tedy následující (3.15).

$$d = \|u\| + o \tag{3.15}$$

Písmenem *s* označíme vzdálenost jednotlivých otisků hrotu štětce od sebe. Nyní můžeme přistoupit k vykreslování. Budeme zjišťovat posunutí hrotu štětce vůči prvnímu bodu přímky. V první iteraci je nutné vzít v úvahu zbytek nevykreslené délky *o* předchozí přímky.

$$\delta_i = \hat{u}_i \cdot (s - o) \tag{3.16}$$

V dalších iteracích se δ vypočítá podle (3.17). Kde *n* je počet iterací a zjistí se vydělením celkové vzdálenosti *d* roztečí *s*.

$$\delta_i = j \cdot \hat{u}_i \cdot s \quad j \in \langle 2, n \rangle \tag{3.17}$$

Souřadnice bodu, na kterém bude ležet střed vykreslené bitmapy, budou tedy (3.18)

$$P_i = (A_i + \delta_i) \tag{3.18}$$

Zbytek vzdálenosti, který se přesune do další přímky bude (3.19).

$$o = d - n \cdot s \tag{3.19}$$

Algoritmus zajistí, že otisky hrotu štětce budou vždy zhruba stejně daleko od sebe. Nebere se v úvahu úhel svíraný přímkami, který může ovlivnit vzdálenost konvových otisků štětce. Zbytek vzdálenosti je započítán, jako by byly přímky rovnoběžné. Vzhledem k tomu, že budeme na přímky rozkládat spojitou křivku, úhly budou malé a nepřesnost můžeme zanedbat.

3.3.2 Aproximace Bézierovy křivky úsečkami

Metoda rasterizace Bézierovy křivky dosazováním parametru *t* do vzorce (3.1) je v [1] popsána jako naivní a neadaptivní, jelikož nezohledňuje velikost zakřivení. Jak bylo řečeno výše, u interpolace bodů popisujících dráhu prstu můžeme předpokládat, že zakřivení nebude velké, takže toto řešení se hodí pro daný problém lépe, než algoritmus de Casteljau, díky nižší výpočetní náročnosti.

Výhodou této metody je možnost předem určit počet úseček, které použijeme k aproximaci křivky. Jako zřejmé nejjednodušší řešení se jeví úprava počtu úseček podle délky křivky. V případech vyžadujících přesnost by bylo výhodné upravovat počet i podle míry zakřivení, nicméně pokud zvolíme dostatečně malou základní délku úsečky, bude to v našem případě zbytečné.

Délka Bézierovy křivky se nejčastěji určuje iterativní metodou, kdy se křivka nejdříve rozdělí na úsečky a pak se sečtou jejich délky. Existuje i analytické řešení [4] ale k jeho spočítání je potřeba hodně aritmetických operací. Pro přibližné přizpůsobení počtu aproximačních úseček stačí hrubý odhad délky křivky. Jelikož je řídící polygon hrubou aproximací tvaru křivky, můžeme použít součet délky přímek mezi řídícími body. Skutečná délka pak bude vždy menší než odhad.

Do výrazu pro výpočet iteračního kroku (3.20) jsem zahrnul ještě rozteč otisků hrotu štětce. Pokud je velká, nemá význam zjišťovat zbytečně velké množství bodů na křivce. Proměnná *s* je rozteč hrotů, *l* je délka křivky a *m* je přibližný požadovaný počet úseček na délku rozteče hrotů. V případě malé délky *l* se výsledek může dostat ven z intervalu <0,1>. Meze je nutné ořezat a zajistit tak použití pouze koncových bodů křivky.

$$\Delta t = \frac{s}{l \cdot m} \tag{3.20}$$

Iterací přes Bézierovu křivku popsanou funkcí (3.1) s krokem Δt dostaneme koncové body úseček tvořících aproximace křivky pro algoritmus z kapitoly 3.3.1.

Dráhu štětce v tomto řešení definuji jako uvolněný uniformní kubický B-sline. Může být vnímána jako posloupnost Bézierových křivek definovaných řídícími body, které musí splňovat podmínky pro spojitost v druhé derivaci.

Pokusně byla implementována i varianta s uniformním kvadratickým B-spline.

3.4 Transformace v lineární algebře a transformační matice

Při návrhu funkcí zvětšení a posunutí plátna vyplynulo, že bude nutné aplikovat na zobrazený obrázek dvě transformace, a to změnu měřítka a posunutí. Máme-li zachovat možnost kreslit na plátno i při aplikované transformaci, bude nutné transformovat zároveň i body z dotykové obrazovky inverzní transformací k transformaci aplikované na plátno.

Afinní transformace vzniká složením několika lineárních transformací. Lze je popsat různě, nejvýhodnější je však popis pomocí matic, který jednak šetří místo a jednak je pohodlný pro výpočet.

3.4.1 Základy lineární algebry a maticového počtu

Množina *L* vektorů se nazývá lineární prostor, pokud jsou na ní definovány operace sčítání a násobení. Sčítání je na množině *L* komutativní a asociativní. Množina *L* musí obsahovat nulový vektor a jednotkový vektor, dále ke každému vektoru musí existovat opačný vektor. Násobení vektoru reálným číslem je asociativní a platí distributivní zákon [5].

Matice je uspořádané schéma reálných čísel. Příklad matice (3.21) je typu *n* x *n*.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \tag{3.21}$$

Vektory z aritmetického lineárního prostoru V^n $\hat{a}_{1i}=(a_{11}, a_{12}, \cdots, a_{1n})$ se nazývají řádkové vektory matice *A*. Vektory z aritmetického lineárního prostoru V^n $\hat{a}_{jl}=(a_{11}, a_{21}, \cdots, a_{nl})$ se nazývají sloupcové vektory matice *A*.

Sčítání matic

Mějme matice *A* a *B* typu m x *n*. Součtem matic *A* a *B* vznikne matice *X* takto:

$$x_{ij}=a_{ij}+b_{ij} \ \forall i, j \in N^+ \tag{3.22}$$

Operace sčítání je definována pouze pro matice se stejnou velikostí řádkových a sloupcových vektorů.

Násobení matic

Mějme matici *A* typu m x *n* a matici *B* typu m x *p*. Matice *X* vzniklá součinem *A* a *B* bude typu m x *p*. Pro prvky matice *X* platí:

$$x_{ij}=\sum_{k=1}^{n} a_{ik} * b_{kj} \tag{3.23}$$

Hodnota na pozici *i,j* v matici *X* tedy odpovídá skalárnímu součinu aritmetických vektorů pro *i*-tý řádek matice *A* a *j*-tý sloupec matice *B*. Součin je definován pouze pokud počet sloupců matice *A* je roven počtu řádků matice *B*.

Je důležité si uvědomit, že nemusí vždy platit *A*B=B*A*.

Lineární transformace

Zobrazení aritmetického lineárního prostoru $V^n \rightarrow V^n$ definované jako $y=C*x$, kde *C* je čtvercová matice řádu *n*, se nazývá lineární transformace ve V^n. Matice *C* se nazývá matice transformace.

Mezi lineární transformace patří otočení, zmenšení/zvětšení a zkosení. V dvou-dimenzionálním prostoru jsou tyto transformace popsány maticí *2* x *2*.

Afinní transformace

Rozšíříme-li lineární transformace o posunutí, dostaneme afinní transformace definované vztahem:

$$w(x)=Ax+B \tag{3.24}$$

V 3.24 představují koeficienty matice *A* lineární transformace a koeficienty matice *B* posunutí. V maticovém zápisu tedy vypadá afinní transformace takto:

$$w: \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{3.25}$$

Maticová reprezentace afinní transformace

Ve vektorové algebře reprezentuje násobení matic vždy lineární transformace a sčítání reprezentuje posunutí. Použitím rozšířené matice je možné reprezentovat afinní transformace násobením jedinou maticí.

Technika vyžaduje rozšíření všech vektorů o jedničku, dále všech matic o řádek nul na spodní straně a o vektor posunutí rozšířený o jedničku v pravém sloupci. Zápis 3.25 upravíme takto:

$$w:\begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix}=\begin{pmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ 0 & 0 & 1 \end{pmatrix}*\begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} \tag{3.26}$$

Tímto se dostáváme vlastně do podmnožiny prostoru rozšířeného o jednu dimenzi. Souřadnice v tomto prostoru jsou příkladem homogenních souřadnic.

3.4.2 Transformační matice pro zvětšení/zmenšení, posunutí, rotaci

Pro usnadnění zápisu se zdrojové vektory uvažují často v řádkové formě, prohodíme tedy v 3.26 řádky a sloupce. Jelikož násobení matic není komutativní, je nutné prohodit i pořadí operandů. Transformovaný vektor bude nyní před transformační maticí.

$$w:[x_{2,}\,y_2,1]=[x_{1,}\,y_1,1]*\begin{pmatrix} a_{11} & a_{21} & 0 \\ a_{12} & a_{22} & 0 \\ b_1 & b_2 & 1 \end{pmatrix} \tag{3.27}$$

Rovnice pro jednotlivé souřadnice vyjdou z násobení matice s vektorem takto:

$$\begin{aligned} x_2&=x_1*a_{11}+y_1*a_{12}+b_1 \\ y_2&=x_1*a_{21}+y_1*a_{22}+b_2 \end{aligned} \tag{3.28}$$

Třetí rovnici není nutné uvažovat, jelikož se v ní v následujících případech vyskytují pouze konstanty. Své uplatnění by si našla například v perspektivním zobrazení.

Nyní uvedu v novém tvaru příklady základních složených transformačních matic, které budou potřeba pro *„zoom/pinch“* funkce plátna programu. Rotace zatím implementována nebude, ale v budoucnu bych ji rád přidal. Problém spočívá v návrhu intuitivní ovládání pro tuto funkci.

Posun bodu o vektor *p*

$$M_p=\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ p_x & p_y & 1 \end{pmatrix} \tag{3.29}$$

Změna měřítka s koeficienty k_x,k_y

$$M_m=\begin{pmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.30}$$

Rotace okolo počátku souřadného systému o úhel α

$$M_r=\begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.31}$$

Změna měřítka o koeficient *k* se středem v bodě *p*[p_x,p_y]

$$M_{mp}=\begin{pmatrix} k & 0 & 0 \\ 0 & k & 0 \\ p_x(1-k) & p_y(1-k) & 1 \end{pmatrix} \tag{3.32}$$

3.4.3 Skládání transformačních matic

Skládání transformační matic se provádí pomocí maticového násobení. Aplikaci matice A na vektor x následovanou aplikací matice B zapíšeme jako:

$$B(A\vec{x}) = (AB)\vec{x} \tag{3.33}$$

Jak bylo uvedeno výše, při násobení matic záleží na pořadí. Máme-li tedy matici A reprezentující posloupnost transformací a matici M reprezentující atomickou transformaci, kterou chceme k posloupnosti přidat, máme dvě možnosti. Můžeme transformaci M aplikovat buď před jak vidíme v 3.34 a nebo po transformacích A jak vidíme v 3.35.

$$A' = MA \tag{3.34}$$

$$A' = AM \tag{3.35}$$

Transformace můžou být obráceny aplikací inverzní matice. Matice A^{-1} tedy odstraní efekt matice A.

4 Charakteristika Android SDK

Android je operační systém postavený na Linuxovém jádře a využívající rozhraní jazyka Java pro vysokoúrovňové programování. SDK poskytuje nástroje jako překladač, debugger a emulátor. Základem pro běh aplikací na zařízení je vlastní virtuální stroj Javy (Dalvik Virtual Machine – DVM).

Vývoj oficiálně spravuje skupina Open Handset Alliance, což je uskupení mobilních operátorů, výrobců telefonů, výrobců polovodičů a softwarových společností. Ve skutečnosti projekt vede Google.

Aplikace v Androidu běží v takzvaném pískovišti (*sandbox*). Každá má vlastní proces pod svým vlastním uživatelským id, které je vygenerováno systémem během instalace. Tato izolace zajišťuje, že špatně navržené nebo škodlivé aplikace nemohou snadno poškodit ostatní.

Podrobný přehled základních znalostí o Androidu a úvod do vývoje včetně mnoha návodů můžeme nalézt například v průběžně aktualizovaném článku Larse Vogela [6].

4.1 Součásti Android aplikace

Aplikace se na platformě Android typicky skládá z těchto hlavních stavebních bloků. Ty které jsou důležité při návrhu kreslící aplikace rozebereme podrobněji později.

- **Activity** představuje prezentační vrstvu aplikace. V podstatě je to obrazovka, kterou uvidí uživatel. Můžeme mít více objektů třídy `Activity` v jedné aplikaci a za běhu je měnit.
- **View** je základní stavební prvek uživatelského rozhraní. Prvky se speciálním chováním se vytváří děděním z `android.view.View`.
- **ViewGroup** tvoří základ pro odvozování tříd, které definují rozložení ovládacích prvků na obrazovce (*layout)*.
- **Service** slouží k provádění operací na pozadí. Mohou informovat uživatele pomocí notification frameworku.
- **ContentProvider** zásobuje aplikaci daty. Může sloužit i ke sdílení mezi různými aplikacemi. Android poskytuje SQLite databázi, která typicky funguje jako zdroj dat.
- **Intent** je asynchronní zpráva, která umožňuje požádat jinou aplikaci nebo službu o provedení nějaké akce. Dělí se na explicitní, kdy žádáme o službu konkrétní aplikaci a implicitní, kdy žádáme systém o seznam aplikací registrovaných k provádění dané služby.
- **BroadcastReceiver** přijímá systémové zprávy a implicitní intenty. Aplikace může zaregistrovat BroadcastReceiver na určité události a systém ji spustí, pokud událost nastane.
- **Widget** je interaktivní komponenta primárně určená k zobrazení na ploše telefonu nebo tabletu. Umožňuje uživateli rychlý přístup k často používaným funkcím nebo informacím.

4.2 Architektura Android aplikace

- **AndroidManifest.xml** slouží k deklaraci komponent aplikace. Všechny Activity a Service třídy, ze kterých se aplikace skládá, tu musí být vyjmenované. Zároveň soubor obsahuje také deklaraci požadovaných povolení k užívání služeb API a hardwaru zařízení.
- **Zdroje (*Resources*)** představují grafiku, rozložení a řetězce použité v aplikaci. Vývojové prostředí automaticky generuje třídu R.java ze souborů umístěných ve složce *res*

v adresářové struktuře projektu. Třída obsahuje id zdrojů jako statické integer hodnoty. S pomocí příslušných SDK metod lze získat jednoduše přímo data. Většina tříd disponuje metodami, které můžeme odkázat jen na id a veškerou práci s dekódováním a použitím zdrojů už obstará systém.

- **Assets.** Zatímco zdroje představují strukturovaná data se specifickým významem a použitím pro platformu, složka *assets* může být použita pro libovolná data. Systém poskytuje třídu *AssetsManager,* usnadňující načítání těchto dat jako vstupní proud.
- **Activity a Layout.** Activity je, jak už bylo řečeno, hlavní obalující prvek grafického uživatelského rozhraní. Představuje obrazovku se specifickým účelem. Uživatelské rozhraní Activity je definováno rozložením (*layout*) objektů rozšiřujících třídu View. Rozložení může být definováno jako XML zdroj, nebo pomocí kódu. Použití XML je preferováno, neboť jasně odděluje aplikační logiku od definice uživatelského rozhraní a navíc umožňuje snadné ladění vzhledu pro různé přístroje.
- **Životní cyklus aplikace** je plně kontrolován systémem. Android může kdykoli pozastavit nebo ukončit aplikaci, například z důvodu příchozího hovoru. Nejdůležitější metody třídy Activity, které je nutné implementovat k zajištění správného chování aplikace, jsou:
  - `onSaveInstanceState()` je voláno pokud je Activity zastavena. Slouží k uložení aktuálního stavu, aby bylo možné po spuštění obnovit práci. Není však vhodné pro zajištění persistence dat, neboť volání není garantováno vždy.
  - `onPause()` je voláno při zmizení Activity. Vhodné pro uvolnění zdrojů a zajištění persistence dat dokumentů.
  - `onResume()` je voláno při spuštění Activity. Vhodné pro inicializaci.
- **Změna konfigurace** je standardně důvodem pro restart Activity. Příkladem takové změny je například otočení zařízení, kdy Activity může chtít načíst jiné rozložení ovládacích prvků. Pokud chceme restartu pro některé události zamezit, je možné to definovat v *AndroidManifest.xml*
- **Třída Context** představuje rozhraní ke globálním informacím a službám systému. Objekt této třídy je nutné udat jako parametr pro většinu systémových volání. Třída Activity je potomkem třídy Context. Je tedy možné ji uvádět v místech, kde je instance Context vyžadována.

4.3 Stavební prvky uživatelského rozhraní a vývoj vlastních komponent

Uživatelské rozhraní aplikace na platformě Android se sestavuje z widgetů, kterým se zde říká *„Views“* a tvoří hierarchické uspořádání. Hierarchie je zajištěna pomocí třídy ViewGroup, která pojme několik widgetů a řídí jejich chování a zobrazování.

4.3.1 View

Základní stavební blok uživatelského rozhraní. Zabírá obdélníkovou oblast na obrazovce a stará se o její vykreslení a správu událostí. Děděním z této třídy se vytváří interaktivní prvky uživatelského rozhraní jako například tlačítka, textová pole a podobně. Tato třída bude základem i pro nejdůležitější vlastní komponentu mé kreslící aplikace, kterou je plátno. Druhou možností, která připadala v úvahu, je použití OpenGL. V takovém případě by vykreslovaní probíhalo na Surface a pravděpodobně by bylo složitější. Dosahovalo by sice vyšší rychlosti, ale rozdíl by nebyl příliš markantní.

Pro vývoj vlastní komponenty je téměř vždy nutné implementovat tyto nejdůležitější metody třídy View:

- `onMeasure (int, int)` je voláno při výpočtu rozložení prvků na obrazovce. View má vrátit svoji požadovanou velikost v daných mezích.
- `onTouchEvent(MotionEvent)` je callback funkce, která vrací objekt obsahující informaci o pohybu prstu po obrazovce.
- `onDraw(Canvas)` je voláno pokud se má část obrazovky, kterou zabírá tento View překreslit. Kreslení se provádí pomocí Canvas objektu, který je předán jako parametr funkce.

4.3.2 ViewGroup

Základní třída, ze které dědíme vlastní *(custom)* rozložení prvků na obrazovce.

Android nabízí několik implementací různých rozložení odvozených od této třídy. Pomocí XML souborů je lze zanořovat a nastavovat. Můžeme tak vytvořit komplexní uživatelská rozhraní. Pokud ale chceme vyšší výkon nebo nestandardní chování, je lepší implementovat vlastní ViewGroup podtřídu.

Je možné i kompletně řídit vykreslování celé ViewGroup a jejích potomků, což se bude hodit při implementaci funkcí „Drag & Drop".

Pro implementaci vlastního rozložení jsou nejdůležitější tyto metody:

- `onMeasure (int, int)` má stejný význam jako u rodičovské třídy View. U ViewGroup je ale nutné provést měření potomků a na základě toho stanovit svou velikost.
- `OnLayout (boolean,int,int,int,int)` je metoda, ve které nastavíme všem potomkům velikost určenou na základě jejich požadavků a dostupného místa.

Pokud chceme upravit vzhled celé ViewGroup, můžeme přepsat metodu `dispatchDraw(Canvas)` třídy View. Ta slouží k vykreslení potomků, ale můžeme ji použít i například k vykreslení pozadí. Rovněž se hodí ke kreslení objektů překrývajících obsah ViewGroup. Zavoláme-li metodu `dispatchDraw(Canvas)` rodiče, máme vykreslené potomky a přes ně můžeme nakreslit například objekt tažený v *Drag & Drop* (*Drag shadow*).

Chceme-li pracovat s událostmi, je důležitá metoda `onInterceptTouchEvent (MotionEvent)`, která nám umožňuje rozhodnout, kdy pošleme události dále do metody `onTouchEvent(MotionEvent)` nebo kdy je předáme přímo potomkům.

4.3.3 Drawable

Třída Drawable představuje abstrakci pro cokoli, co může být vykresleno. Nejčastěji představuje obal pro grafické zdroje definované v XML. Můžeme však libovolně programově definovat vykreslení plochy Drawable a objekty vzniklé třídy používat jako pozadí nebo obsah knihovních widgetů.

Na rozdíl od třídy View nemá Drawable žádný mechanizmus pro zpracování událostí nebo interakci s klientem. Představuje pouze grafický objekt.

Drawable má navíc stav. To umožňuje snadno měnit vhled grafiky podle stavu widgetu, jehož vzhled Drawable reprezentuje.

Drawable podporuje animaci přes rozhraní Drawable.Callback. Klient by měl zaregistrovat implementaci rozhraní přes metodu `setCallback(Drawable.Callback)`, aby animace fungovala. Systémové prostředky jako metoda `setBackgroundDrawable(Drawable)` toto rozhraní podporují a implementace animovaných widgetů je tedy velmi snadná.

Nejdůležitější metody třídy a metody, které budeme nejčastěji implementovat jsou:

- `setBounds(Rect)` musí být zavoláno před vykreslením, aby bylo možné určit rozměr požadované grafiky.
- `getPadding(Rect)` vrací oblast určenou pro obsah. Například textové tlačítko, které má Drawable jako pozadí, si podle vráceného obdélníku vycentruje popisek.
- `draw(Canvas)` je metoda, ve které definujeme vzhled vlastní drawable. Slouží k vykreslení obsahu do oblasti ohraničené obdélníkem nastaveným v `setBounds(Rect)`. Metoda by měla respektovat volitelnou průhlednost a barevný filtr, k jejichž nastavení má třída připraveny abstraktní metody.

4.3.4 Dialog

Dialog je okno, které se zobrazí přes obsah současné Activity a převezme uživatelský vstup. Activity nabízí nástroje pro řízení životního cyklu dialogů přes metody `onCreateDialog(int)`, `onPrepareDialog(int,Dialog)`, `showDialog(int)` a `dismissDialog(int)`.

Pro vývoj vlastního dialogu je důležitá metoda `setContentView(View)`, která stejně jako u Activity nastaví obsah okna. Většinou použijeme rozložení widgetů definované v XML.

Dále je důležitý konstruktor `Dialog(Context,int)`. Integer parametr umožňuje definovat id stylu a změnit tak vzhled a chování dialogu.

4.3.5 Styles.xml

Jedná se o soubor definice zdrojů umístěný v adresáři *res*. Definuje vlastní styly, které umožňují ovládat vhled systémových prvků jako jsou Dialog a Activity. K id stylu se dostaneme v kódu přes `R.styles`.

4.3.6 PopupWindow

PopupWindow umožňuje zobrazit libovolný View v plovoucím kontejneru na vrcholu Activity. Může být dobrou alternativou k Dialogu, pokud nechceme přesměrovat všechen uživatelský vstup do nového okna.

4.4 Grafické a jiné důležité pomocné třídy

Vyvíjená aplikace bude stát převážně na grafických nástrojích Android SDK. Zde popíši knihovní třídy, které budou tvořit základ pro vývoj plátna ke kreslení prstem.

4.4.1 Bitmap

Bitmap je třída obalující nativní kód pracující na Androidu s bitmapami. Bitmapa představuje obrázek uložený v paměti. Pro většinu aplikací bude převážná část grafiky představovat neměnné grafické zdroje, které se však mohou objevovat v kódu na různých místech. Proto Android SDK obsahuje optimalizace zabraňující výskytu redundantních bitmap v paměti.

Nové objekty třídy Bitmap lze vytvářet pouze přes statické tovární metody třídy Bitmap nebo přes tovární metody BitmapFactory, což je třída obsahující pouze statické metody a starající se o dekódování bitmap z různých zdrojů.

Pokud vytváříme bitmapu z grafického zdroje (*resource*), bude neměnná (*immutable*). Díky této optimalizaci můžou být stejná pixelová data použita pro více instancí třídy bitmap. Stejná optimalizace bude použita i pokud vytváříme kopii jakékoli neměnné bitmapy.

Celkově jde jistě o přínos, ale neměnné bitmapy přináší i problémy. Ve starších verzích systému chybí v knihovně možnost vytvořit ze zdroje nebo proudu dat přímo měnitelnou bitmapu. Je nutné nejdříve dekódovat neměnnou bitmapu a pak vytvořit kopii, které nastavíme možnost editace. Tento postup však vyžaduje mít v jeden okamžik v paměti data dvakrát. Vhledem k tomu, že aplikace má na androidu znatelně omezené množství paměti k využití, vniká u větších obrázků v paměťově náročných aplikacích v podstatě neřešitelný problém. Jedinou možností pak zůstává snížit rozlišení tak, aby dostupná paměť vystačila.

Nejdůležitější metody třídy bitmap pro mnou vyvíjenou aplikaci jsou následující:

- `Bitmap createBitmap(int,int,Bitmap.Config)` vytvoří novou bitmapu o zadané velikosti. Tuto bitmapu lze měnit, takže do ní lze kreslit přes Canvas objekt. Enum Bitmap.Config slouží ke specifikaci typu bitmapy. Můžeme pomocí něj ovlivnit počet bitů, kterým se budou kódovat barevné složky, a tím i kvalitu obrazu. Speciální možností je bitmapa obsahující jen alfa kanál. Hodí se například na vytváření různých masek.
- `Bitmap copy(Bitmap.Config, boolean)` je nejschůdnější metodou, jak převést neměnnou bitmapu na editovatelnou. Dosáhneme toho nastavením druhého parametru metody na *true.*
- `compress(Bitmap.CompressFormat, int, OutputStream)` je velice pohodlná metoda k ukládání obrázků do formátů jpeg a png. Výsledek je zapsán do výstupního proudu, takže ho můžeme uložit kamkoli potřebujeme, včetně například databáze.
- `recycle()` uvolní nativní bitmapový objekt svázaný s tímto Bitmap objektem. Paměť je dostupná hned a nemusí se čekat na *Garbage collector*. To se hodí zejména potřebujeme-li nahradit velký obrázek novým velkým obrázkem.

4.4.2 Canvas

Canvas slouží na Androidu k vykreslování 2D grafiky z kódu. Obaluje volání, jejichž pomocí systém vykresluje grafická primitiva. Abychom mohli něco vykreslit, potřebujeme obecně čtyři základní komponenty. Bitmapu, která obsahuje jednotlivé pixely, Canvas, který umožňuje volání funkcí vykreslování, vykreslované primitivum (geometrická primitiva, texty, bitmapy) a Paint objekt, který popisuje barvy, styly, tloušťky čar a další parametry vykreslování.

Volání, která třída canvas umožňuje, tu kvůli rozsahu nebudu popisovat, lze je najít v [7].

4.4.3 Paint

Paint je třída, jejíž instance nesou informace o parametrech vykreslování třídou Canvas. Nejčastěji ho uvidíme nastavovat barvu vykreslování, ale obsahuje i komplikovanější funkce. Například shadery umožňují nahradit barvu vykreslovaného objektu gradientem.

4.4.4 Matrix

Matrix obsahuje transformační matici typu 3x3. Metody této třídy implementují operace popsané v 3.4. Metody s předponou *set* nastaví matici na danou transformaci. Metody s předponou *pre* provedou násobení matice danou transformací tak, že nová transformace bude před původní

(viz 3.34). V případě metod s předponou *post* bude naopak nová transformace následovat po původních transformacích (viz 3.35).

4.4.5 MotionEvent

MotionEvent je třída, v jejíchž objektech jsou předávány informace o pohybu prstů po dotykové obrazovce.

Základními informacemi v objektu je pozice události na obrazovce a kód akce. Jeden objekt může obsahovat i více pozičních bodů. Potom poslední pozici získáme metodami `getX` a `getY`. Předchozí body metodami `getHistoricalX` a `getHistoricalY`. Počet historických bodů zjistíme metodou `getHistorySize`.

Na obrazovce může být najednou více prstů. Pozice každého z nich se nazývá ukazatel (*pointer*) a má svůj index a id. Prsty můžeme na obrazovku pokládat a zvedat v libovolném pořadí. Index ukazatele vždy začíná od nuly a postupuje výše, takže se může mezi jednotlivými událostmi měnit. Naproti tomu id zůstává konstantní, dokud je ukazatel aktivní. Id můžeme zjistit z indexu metodou `getPointerId`.

Kódy akcí získáme metodou `getAction` a jsou následující:

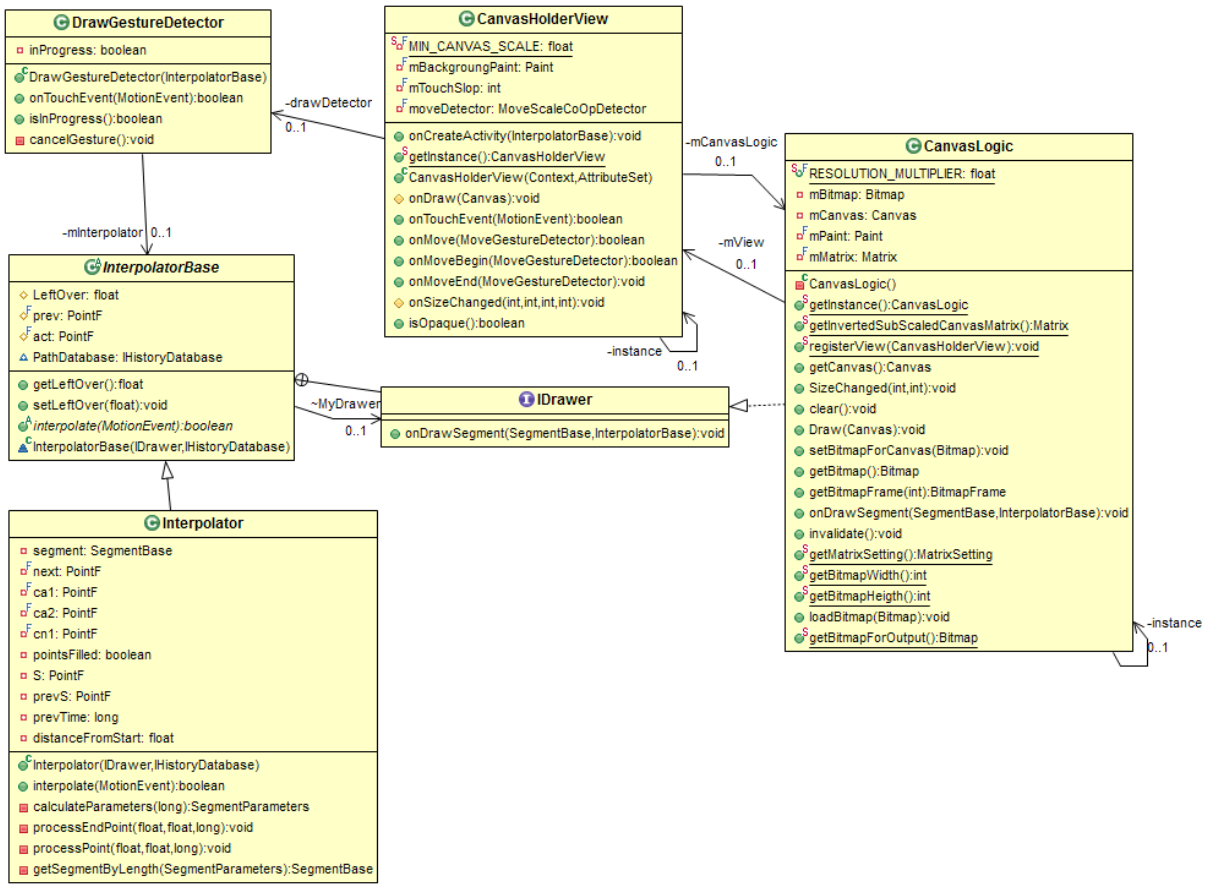
- **ACTION_DOWN** – První prst se dotkl obrazovky. Byl aktivován primární ukazatel.
- **ACTION_POINTER_DOWN** – Neprimární ukazatel byl aktivován. Nějaký další prst se dotkl obrazovky.
- **ACTION_POINTER_UP** – Neprimární ukazatel byl deaktivován.
- **ACTION_UP** – Primární ukazatel byl deaktivován. Poslední prst se zvedl z obrazovky.
- **ACTION_CANCEL** – Aktuální gesto bylo zrušeno.
- **ACTION_MOVE** – Došlo k pohybu ukazatelů po obrazovce. Vždy nastane mezi ACTION_DOWN a ACTION_UP.

5 Implementace základních funkcí

Tato kapitola se bude zabývat implementací a využitím výše uvedených algoritmů štětce a interpolátoru ke kreslení na plátno aplikace. Transformační matice budou použity k implementaci funkcí posunutí a zvětšení plátna.

5.1 Interpolace a vykreslování segmentů

Základ programu tvoří kreslící plátno reprezentované bitmapou. Tato bitmapa je vykreslována na obrazovku zařízení pomocí třídy CanvasHolderView, která je potomkem View. Tato třída se stará také o sběr pohybových událostí, které pak předává interpolátoru.



Obrázek 15: Diagram tříd interpolace a vykreslováni

Třída CanvasHolderView obsahuje komponentu DrawGestureDetector. Ta se stará o rozpoznávání gesta kreslení a filtruje události, které patří k jiným gestům, aby nedocházelo k nechtěným čárám například při zoomu.

DrawGestureDetector má pole s konkrétní instancí abstraktní třídy InterpolatorBase, které předává pohybové události rozpoznané jako gesta kresby. Abstraktní bázová třída tu měla význam zejména kvůli testování a porovnáváni různých implementací interpolátoru.

Interpolátor z přijatých pohybových událostí vytváří segmenty stopy (přímky, křivky). Ty jsou pak posílány k vykreslení na plátno. Komunikaci interpolátoru s logikou plátna zajišťuje rozhraní IDrawer pomocí *callback* metody `onDrawSegment`.

Logika plátna (CanvasLogic) zajišťuje operace spojené s vykreslováním. Třída CanvasHolderView tím, že se stará o vykreslení bitmapy plátna na obrazovku, tvoří jakousi obálku logiky plátna vzhledem k Android frameworku.

5.1.1 Interpolátor

Během vývoje byly pokusně implementovány tři verze interpolátoru využívající přímkové, kvadratické a kubické segmenty. Ve výsledném programu je použita verze s kubickými segmenty, jelikož křivky spojité v druhé derivaci vypadají nejvíce hladce.

Nejdůležitější metodou interpolátoru je `interpolate(MotionEvent)`. Zde se dělí pohybové události podle toho, zda se jedná o první, poslední nebo jiný bod tahu. První bod inicializuje algoritmus, poslední ho ukončuje. Ostatní body jsou zpracovány v metodě `processPoint()`. Tato metoda dopočítává body spline křivky tak, aby platili podmínky spojitosti z kapitoly o spojování Bézierových křivek (3.2.2). Abychom určili tvar segmentu, musíme vědět, kde bude ležet následující bod splinu. Výpočet tedy probíhá se zpožděním jednoho bodu.

Pro kubický interpolátor by metoda vypadala takto:

```
private void processPoint(float x, float y) {
      next.x = x;
      next.y = y;

      ca2.x = 1f/3f*prev.x + 2f/3f*act.x;
      ca2.y = 1f/3f*prev.y + 2f/3f*act.y;
      cn1.x = 2f/3f*act.x + 1f/3f*next.x;
      cn1.y = 2f/3f*act.y + 1f/3f*next.y;

      S = new PointF();
      S.x = (ca2.x + cn1.x)/2;
      S.y = (ca2.y + cn1.y)/2;

      segment = new Cubic(prevS,S,ca1,ca2);
      MyDrawer.onDrawSegment(segment, this);
      PathDatabase.storeSegment(segment);

      //***Prepare next iteration**********
      prevS = S;
      prev.x = act.x;
      prev.y = act.y;
      act.x = next.x;
      act.y = next.y;
      ca1.x = cn1.x;
      ca1.y = cn1.y;
}
```

Body *cax* jsou řídící body právě počítaného segmentu, body *cnx* jsou řídící body následujícího segmentu. Body *prev*, *act* a *next* jsou řídicí body spline křivky. Body *prevS* a *S* jsou koncové body segmentu. Podle pravidla z 3.2.3 rozdělíme přímku mezi kontrolními body splinu na třetiny řídícími

body segmentu. Na středu přímky mezi posledním kontrolním bodem aktuálního segmentu a prvním kontrolním bodem nového segmentu pak bude ležet koncový bod.

Pro kvadratický spline by byl interpolátor o něco jednodušší. Vypadal by takto:

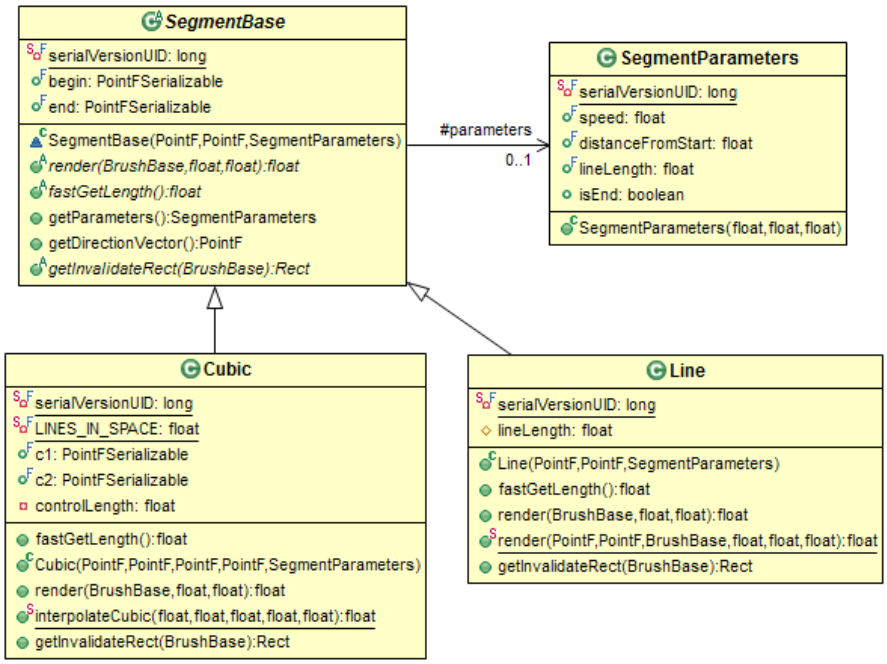
```
private void processPoint(float x,float y){
      act.x = (x + mX)/2;
      act.y = (y + mY)/2;
      curve = new Quad(new PointF(prev.x,prev.y),
            new PointF(act.x,act.y),new PointF(mX,mY));

      MyDrawer.onDrawSegment(curve, this);

      prev.x = act.x;
      prev.y = act.y;
      mX = x;
      mY = y;
}
```

Kvadratický interpolátor vychází z pravidla spojitosti v první derivaci, koncový bod musí ležet na středu přímky tvořené řídícími body. Body z dotykových událostí tvoří vlastně řídící body Bézierových křivek. Koncové body pak dopočítáváme tak, aby bylo splněno pravidlo spojitosti.

Interpolátor využívající přímky pouze spojoval body, takže to vlastně žádný interpolátor nebyl. Sloužil pouze k ověření funkčnosti architektury.

5.1.2 Segmenty stopy štětce

Segmenty stopy štětce tvoří Bézierovy kubiky nebo přímky. Bézierovy křivky skládají spline vypočítaný interpolátorem. Pokud jsou však koncové body křivky tak blízko u sebe, že by na několika málo pixelech tvar křivky stejně nevynikl, je segment z důvodu optimalizace nahrazen přímkou.



Obrázek 16: Diagram tříd segmentů stopy štětce

Každý segment se musí umět vyrenderovat metodou `render (BrushBase, float, float)`. Využívá k tomu metodu `stampAt` štětce v prvním parametru. Dále potřebuje znát vzdálenost otisků od sebe. Tuto informaci zjistí rovněž ze štětce. Poslední parametr je aktuální zvětšení plátna. Pokud není koeficient zvětšení jedna, je nutné upravit vzdálenost otisků, aby se pak ve výsledné bitmapě jevily vždy stejně daleko.

K renderování segmentů úseček je použit algoritmus popsaný v 3.3.1. Křivku je nejdříve nutné rozložit na malé úsečky. Pak je rovněž použit stejný algoritmus.

Rozklad se provede dosazováním řídící proměnné v rozsahu od nuly do jedné do rovnice 3.4. Tuto rovnici jsem implementoval s ohledem na minimální počet nutných aritmetických operací takto:

```
public static float interpolateCubic(float u, float p0, float p1, float
p2, float p3) {
        float oneMinusU = 1.0f - u;
        float oneMinusU2 = oneMinusU * oneMinusU;
        float u2 = u * u;
        return p0 * oneMinusU2 * oneMinusU +
                    3.0f * p1 * u * oneMinusU2 +
                    3.0f * p2 * u2 * oneMinusU +
                    p3 * u2 * u;
}
```

Výpočet se provádí pro x-ovou a y-ovou souřadnici zvlášť.

Z hlediska výkonu metody je důležité rozhodnutí, na kolik přímek bude před renderováním křivkový segment rozdělen. Rozhodl jsem se navrhnout řešení, které bere v úvahu přibližnou délku segmentu. Délka kroku se stanovuje takto:

$$step = \frac{spacing}{length * linesInSpace} \qquad (5.1)$$

Hodnota *linesInSpace* udává počet přímek segmentu na vzdálenost otisků hrotu štětce. Na první pohled to vypadá, že by tato hodnota mohla být prostě jedna, jelikož bitmapa bude vždy otisknuta až ve vzdálenosti rovné *spacing*. Musíme ale brát v úvahu, že otiskování nezačíná od začátku segmentu, nýbrž je posunuté o vzdálenost, které zbyla z renderování předchozího segmentu (*leftOver*). Díky tomu nebudou otisky ležet v koncových bodech vypočtených interpolací Bézierovy křivky, ale libovolně na úsečce tvořené těmito body. Větším počtem úseček tedy snížíme odchylku od reálné pozice na křivce. Nicméně příliš velké množství už nepřinese žádné znatelné zlepšení. Hodnotu konstanty počtu úseček na rozteč hrotů štětce jsem stanovil pokusně na čtyři.

Pokud by výše uvedená rovnice vrátila například pro krátký segment hodnotu vyšší než 0,5 bude použito 0,5. Méně než dvě přímky na segment by už nefungovaly dobře.

Dále se vynásobením vypočteného kroku s odhadovanou délkou segmentu zkontroluje, jestli jsou výsledné úsečky delší než jeden pixel. Menší hodnota by totiž neměla smysl, jelikož by se nijak neprojevila větší kvalitou.

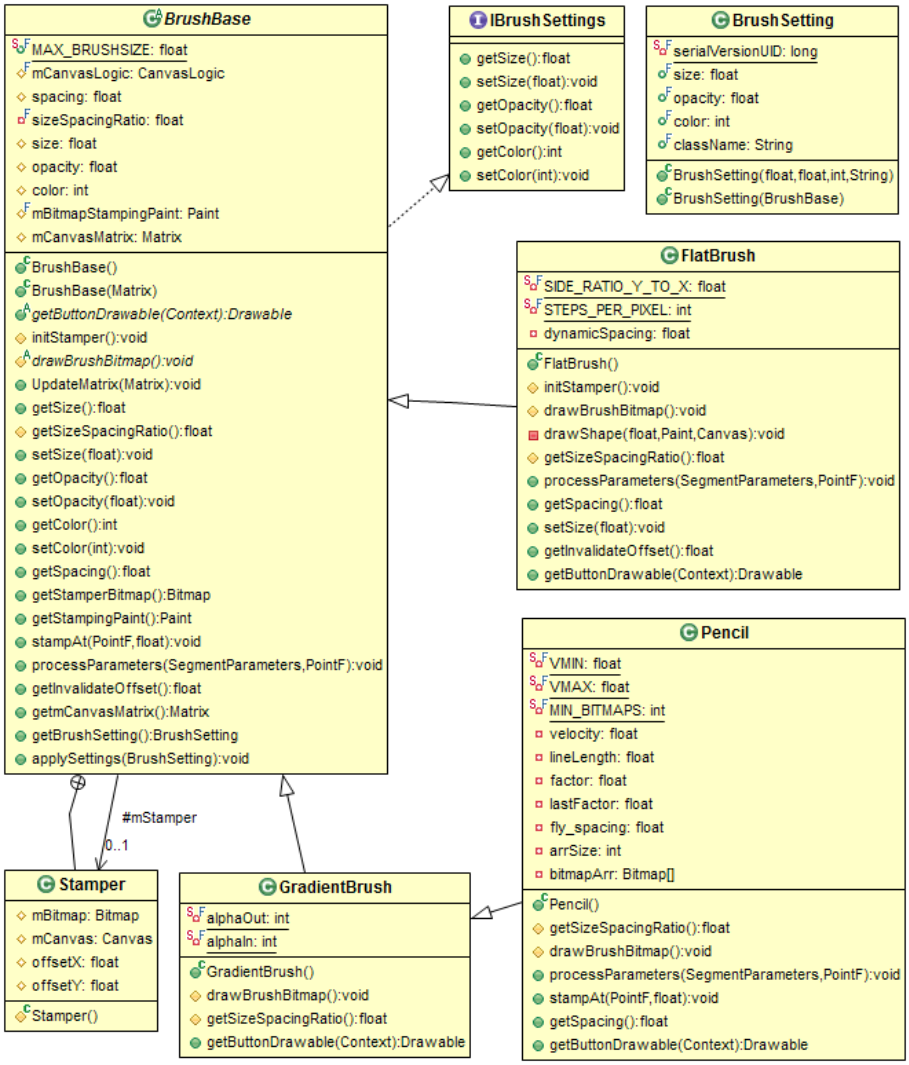
Nakonec provedeme vlastní renderování. Postupně v cyklu navyšujeme hodnotu řídící proměnné o velikost kroku a pomocí výše uvedeného fragmentu kódu získáváme souřadnice koncových bodů úseček. Úsečky jsou pak renderovány algoritmem z kapitoly 3.3.1. Implementaci tohoto algoritmu je možné nalézt v příloze 1.

Délka se v případě kubického segmentu aproximuje jako součet délek úseček mezi kontrolními body křivky.

Obalový obdélník je určen podle nejlevějšího, nejpravějšího, nejvrchnějšího a nejspodnějšího bodu.

5.2 Štětce

Interpolátor vytvoří segmenty křivky a pošle je na vykreslení. Než však něco vykreslíme, potřebujeme pro algoritmus tahu štětcem ještě další parametry. Bitmapu, kterou budeme otiskovat, a pak vzdálenost jednotlivých otisků od sebe (*spacing*). Třídy štětců kontrolují tuto hodnotu a starají se o přípravu bitmapy hrotu štětce ve správné velikosti. Pokročilejší štětce mohou navíc řídit parametry během tahu na základě různých veličin. Například rychlosti prstu.



Obrázek 17: Diagram tříd štětců

Rozhraní IBrushSettings definuje vlastnosti společné pro všechny štětce. Jedná se o velikost, průhlednost a barvu. Abstraktní bázová třída pro štětce BrushBase toto rozhraní implementuje. Všechny štětce tedy musí podporovat toto nastavení.

Třída BrushBase obsahuje razítko (*stamper*). Jedná se o třídu, která seskupuje aktuální bitmapu hrotu štětce s offsetem, o který je při otiskování posunut levý horní bod bitmapy od bodu na křivce, kam se má hrot otisknout. Razítko dále obsahuje odkaz na aktuální plátno, takže se při vykreslování pouze zavolá metoda `stampAt(PointF,float)`.

Druhý parametr výše uvedené metody umožňuje třídě štětce vědět, jak daleko se nachází v právě vykreslovaném segmentu. Spolu s metodou `processParameters (SegmentParameters, PointF)` umožňuje tvorbu štětců schopných v závislosti na parametrech měnit otiskovanou bitmapu. Parametry této metody tvoří objekt nesoucí informace o různých vlastnostech segmentu a vektor určující směr segmentu podle koncových bodů.

Třída BrushSetting slouží u uchování informací, podle kterých je možné obnovit perzistentní nastavení štětce. Používá ji například historie (Dědí z třídy HistoryFrame, takže je možné objekty vkládat přímo do zásobníku historie). Při ukládání je využito rozhraní Serializable.

Zatím jsem implementoval následující štětce. V budoucnu by se jejich počet měl zvětšit a rád bych implementoval různé pokusné štětce využívající například i senzorů telefonu a podobně.

- **GradientBrush** je základní štětec, jež je při krajích průhledný a směrem do středu přibývá na plnosti. K vykreslení bitmapy je použit shader RadialGradient. Vykreslování je takto rychlejší a výsledek má dokonalou kvalitu bez artefaktů.
- **Pencil** vychází z GradientBrush. Přidává navíc dynamickou změnu velikosti a průhlednosti v závislosti na rychlosti tahu prstem. Navíc na začátku a konci čáry v prvním a posledním segmentu zmenšuje postupně bitmapu a tvoří tak pěknou špičku. Podobnou jako uvidíme na papíře uděláme-li rychlý tah tužkou.
- **FlatBrush** je plochý štětec. Bitmapa je vykreslována postupným kreslením průhledných oválů přes sebe. Velikost oválu se v každém kroku zmenší v obou směrech o jeden pixel. Nakonec vznikne ve středu přímka. Pro lepší vhled je navíc nutné upravovat rozteč otisků bitmapy v závislosti na směru tahu. Pokud je tah veden směrem, kde je štětec užší, musí být vzdálenost otisků od sebe menší a naopak.

Implementoval jsem ještě jeden štětec umožňující vykreslení bitmapy hrotu štětce s volitelnou šířkou hladkého okraje. Vykreslovaní se provádělo skládáním průhledných kružnic s odstupňovanou velikostí na sebe. Metoda však nedosahovala takové kvality, jako gradient vykreslený systémovou funkcí. Zejména u malých bitmap byly značné artefakty, které se pak promítaly i na vzhled čáry.

5.3 Transformační matice pro zvětšení a posunutí plátna

Vykreslení plátna na obrazovku zařízení probíhá v metodě `onDraw` třídy CanvasHolderView. Tato metoda volá funkci `Draw` třídy CanvasLogic. Plátno představuje bitmapa, do které se provádí veškeré vykreslování. Zobrazení na ploše prvku View se provede vykreslením na plátno dodané jako parametr metodou `onDraw`. Plátno má mimo jiných metod k vykreslování bitmap i metodu `drawBitmap (Bitmap, Matrix, Paint)`, která při vykreslování transformuje zadanou maticí. K posunutí nebo změně velikosti plátna stačí tedy vytvořit matici s požadovanou transformací a v této metodě ji použít. Zároveň je nutné vykreslit nejdříve pozadí. Jinak by zůstalo bílé, což je i výchozí barva plátna, takže by nebylo vidět, kde se nachází.

Kreslící program používá transformace posunutí a změny velikosti (*scale,translate*). Transformace jsou aplikovány na matici obsahující už existující transformace. Musíme tedy používat metody s předponou *post*.

S transformační maticí pracuje mnoho dalších modulů programu. Proto jsem ji udělal globálně dostupnou přes statické metody třídy MatrixLogic. Tato třída zároveň umožňuje pohodlně provádět nad maticí potřebné operace.

Máme-li na plátně aplikovanou transformaci, musíme počítat s tím, že body z pohybových událostí už nebudou pozicí odpovídat souřadnicím v bitmapě, kam se mají vykreslit. Abychom toto zkorigovali, musíme souřadnice bodů transformovat inverzní maticí k té, kterou jsme použili pro transformaci bitmapy plátna.

Transformace se provádí při otiskování v metodě `stampAt`. Štětec má k dispozici instanci inverzní matice, která je po každé změně aktualizována. Jelikož se do historie ukládají segmenty tak, jak je zpracoval interpolátor z pohybových událostí, musíme sledovat a uchovávat i změny inverzní transformační matice. Při renderování historie jsou pak opět matice použity štětci.

5.4 Detektory gest

Platforma Android nabízí třídu GestureDetector a ScaleGestureDetector pro usnadnění práce s pohybovými událostmi. Tyto třídy přijímají pohybové události (metoda `onTouchEvent`) a pokud v nich rozpoznají sémantiku nějakého gesta, vyvolají callback funkci, kde je možné implementovat reakci na událost.

Rozhodl jsem se přijmout tento přístup a implementovat podobné třídy i pro gesta, se kterými bude pracovat kreslící program.

5.4.1 DrawGestureDetector

DrawGestureDetector je velmi jednoduchý a slouží především k odfiltrování událostí, které patří jiným detektorům dřív, než jsou předány dále ke zpracování. Gesto kreslení je definováno jako tah jedním prstem. To znamená, že musí být aktivní pouze jeden primární ukazatel. Jakmile je aktivován další, gesto je zrušeno. Začátek gesta se bere až po pohybu (kód ACTION_MOVE).

5.4.2 MoveGestureDetector

Tato třída slouží k detekci gesta pro posouvání plátna. Posunutí je často prováděno společně se změnou velikosti, proto je dobré navrhnout gesto tak, aby se dalo provádět co nejrychleji po nebo současně s tímto gestem. Na to jsem bral ohled a navrhl gesto jako pohyb dvou prstů po obrazovce, přičemž vzdálenost mezi nimi zůstává na rozdíl od zvětšování a zmenšování konstantní (Stejný přístup často používají i jiné programy, takže by měl být pro uživatele intuitivní).

Detektor rozpozná gesto, pokud jsou aktivní dva ukazatele. Z pohybu ukazatelů mezi jednotlivými událostmi vytvoří vektory. Plátno je pak posouváno o delší ze dvou vektorů.

5.4.3 MoveScaleCoOpDetector

Tato třída rozšiřuje schopnosti MoveGestureDetectoru o rozpoznávání a výpočet parametrů gesta změny velikosti. Používá k tomu třídu Android frameworku ScaleGestureDetector.

Toto gesto se prování dvěma prsty. Pohyb špiček od sebe nebo k sobě udává faktor změny velikosti. Dalším parametrem je pak ohniskový bod.

Ke správnému chování posouvání a změny velikosti plátna je nutné zajistit vzájemné vyloučení obou gest. Gesto posunu má prioritu. Pokud je aktivní, jsou všechny faktory změny velikosti odmítány a akumulovány dokud posun neskončí.

Při detekci pohybu se vychází z předpokladu, že při posunu prstů na opačnou stranu se budou vektory odčítat a při posunu stejným směrem naopak sčítat. Pro aktivaci gesta musí být součet vektorů alespoň jeden a půl násobek delšího vektoru. Můžeme se na to také dívat tak, že druhý prst musí urazit alespoň polovinu vzdálenosti stejným směrem jako první prst.

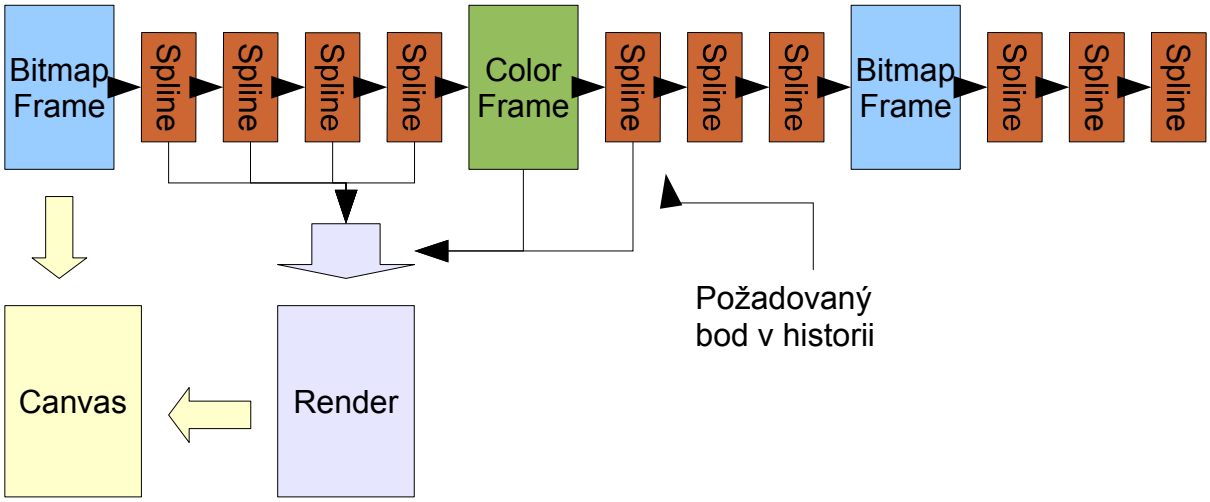
MoveScaleCoOpDetector připravuje parametry pro aplikaci transformací na globální transformační matici uloženou v MatrixLogic.

5.5 Historie akcí

Historie akcí má sloužit především k implementaci funkcí krok zpět a krok vpřed. Nicméně u některých kreslících programů jsem si všiml zajímavé funkce umožňující zrychlené přehrávání průběhu kresby. Rozhodl jsem se navrhnout historii tak, aby jednak umožňovala vracet zpět

provedené akce a jednak byla připravená jako zdroj dat, pokud se v budoucnu rozhodnu implementovat přehrávání kresby.

Na obrázku 18 je blokové schéma historie akcí. Data jsou ukládána ve formě posloupnosti vektorových reprezentací spline křivek popisujících jednotlivé tahy prstem při kresbě. Mezi ně jsou pak vkládány bitmapové klíčové snímky (*key frames*), vůči kterým se renderují vektorová data. Dále jsou vkládány rámce popisující změnu nastavení modulu vykreslování. Například změnu barvy, změnu transformační matice, změnu nastavení štětce.



Obrázek 18: Blokové schéma historie akcí

Při návratu do libovolného bodu v historii probíhá vykreslování následujícím způsobem. Nejbližší starší klíčový snímek je použit jako bitmapa plátna. Dále musíme nalézt nejbližší starší rámec barvy, transformační matice a štětce, abychom uvedli modul vykreslování do správného výchozího stavu. Potom postupujeme od klíčového snímku posloupností rámců vpřed. Pokud narazíme na rámec spline křivky, vykreslíme ho jeho metodou `render (BrushBase, float, float)` s použitím štětce z nejbližšího předchozího rámce v prvním parametru metody. Pokud narazíme na jakýkoli jiný rámec, změníme příslušné nastavení renderovacího modulu.

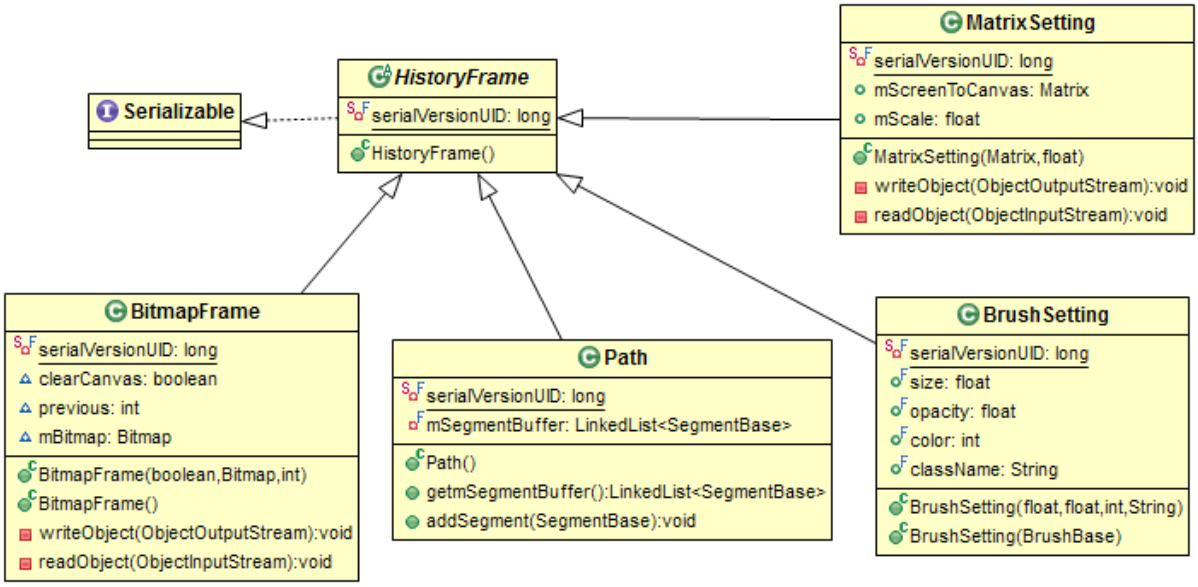
Tento systém je kompromisem mezi paměťovou náročností a rychlostí renderování při návratu v historii. Pokud bychom nevkládali bitmapové rámce, musela by se při každém kroku vyrenderovat všechna vektorová data od začátku kresby na prázdné plátno. To by trvalo neúměrně dlouho i na stolním počítači, natož na mobilním zařízení. Chceme-li dosáhnout dobré uživatelské zkušenosti, musíme dobu čekání co nejvíce zkrátit. Na druhé straně bitmapový snímek ukládá celé plátno, které může být poměrně velké (zatím je jeho velikost omezena dostupnou pamětí přidělenou aplikaci systémem Android). Pohybuje se řádově ve stovkách kilobajtů po kompresi. Otázkou je, jak správně stanovit frekvenci vkládání klíčových snímků tak, aby nebyly zbytečně vysoké nároky na paměť a zároveň se dosáhlo rozumné rychlosti vykreslování.

Pokusně jsem stanovil frekvenci vkládání klíčových snímků jednou na deset snímků s vektorovými spline daty. Rychlost vykreslování je na mém zařízení (Dell Streak, Qualcomm Snapdragon 1GHz procesor, žádná hardwarová grafická akcelerace) přijatelná, ale nepříjemně se zhoršuje pokud jsou renderovány velmi dlouhé čáry.

Tomuto jevu by měla zamezit úprava závislosti frekvence vkládání klíčových snímků z počtu křivek na délku křivek. Potom bude počet otisků hrotu štětce mezi jednotlivými snímky zhruba

konstantní a nemělo by docházet k extrémním případům, kdy se doba vykreslování protáhne silně nad průměr.

Dobu vykreslování částečně ovlivňuje i velikost bitmapy hrotu štětce. Nicméně tyto drobné výkyvy už jsem se rozhodl zanedbat.

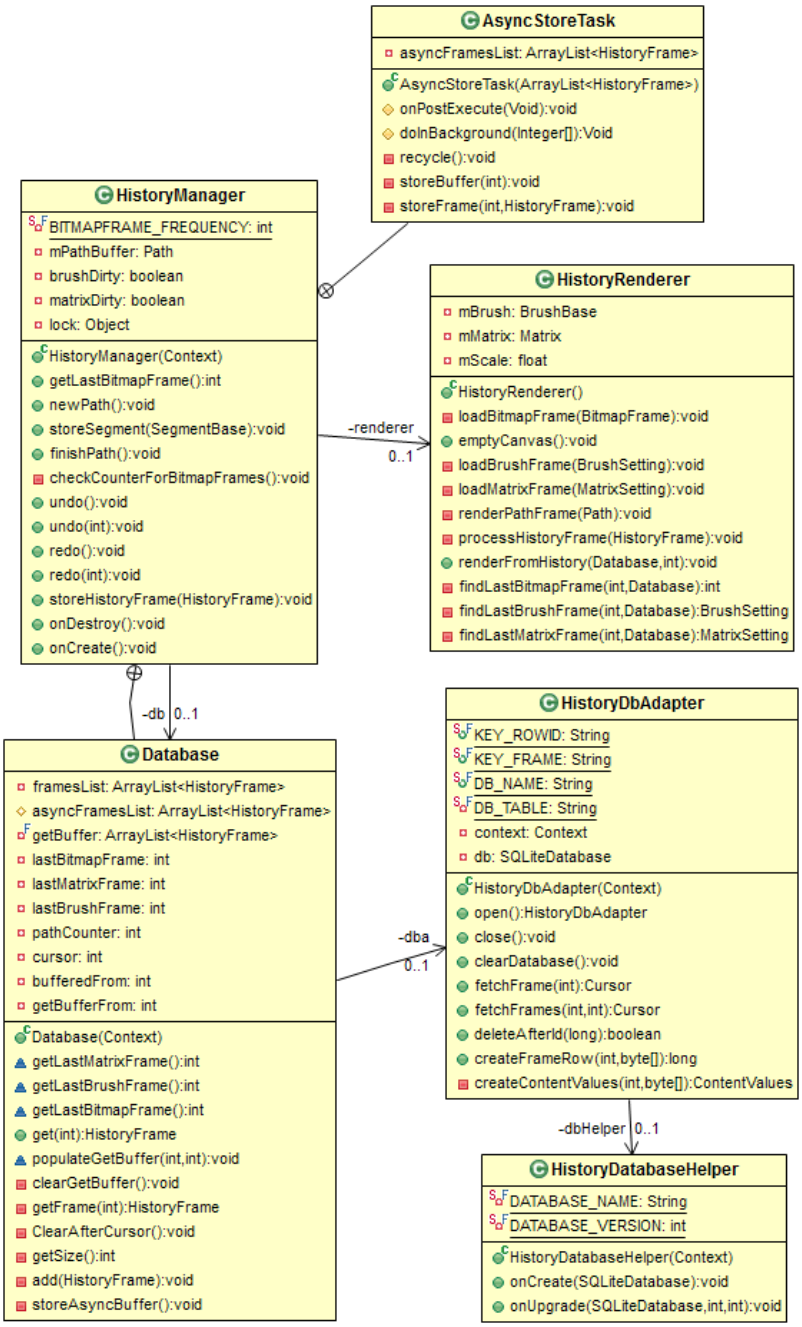


Obrázek 19: Diagram tříd rámců historie

Na obrázku 19 vidíme diagram tříd rámců historie. Potomci třídy HistoryFrame musí implementovat rozhraní Serializable, protože tento Java mechanizmus je použit při serializaci objektů pro ukládání do databáze.

Třída BitmapFrame představuje klíčový snímek. Obsahuje buď bitmapu plátna, nebo povel k vymazání všeho. Při serializaci musí třída zkomprimovat obsaženou bitmapu pomocí metody `Bitmap.compress()` do bufferu. Ten je pak uložen jako pole třídy v databázi. Při deserializaci musí zase komprimovaný buffer dekódovat na bitmapu. Zde nastává navíc výše zmíněný problém s neměnností bitmapy. Pokud ji chceme použít pro plátno, musíme vytvořit editovatelnou kopii. Nutnost počítat s místem pro kopii bitmapy omezuje velikost plátna, na kterou vystačíme s pamětí.

Třída Path představuje spline křivku. Tvoří ji seznam segmentů od interpolátoru. MatrixSetting obsahuje transformační matici a údaj o aplikovaném měřítku. BrushSetting obsahuje parametry štětce a název třídy konkrétního nástroje. Při deserializaci je podle plně kvantifikovaného jména vytvořen objekt a poté jsou mu nastaveny parametry podle polí BrushSetting.



Obrázek 20: Diagram tříd správy a renderování historie

5.5.1 HistoryManager

Tato třída řídí práci s historií. Bufferuje nakreslené čáry, klíčové snímky a rámce s nastavením. Ukládá tato data do databáze a pak je z ní čte. Také se stará o vykreslení při požadavku na návrat do určitého bodu historie.

Database

Je vnitřní třídou (*inner class*) HistoryManageru. Obsahuje buffer, do kterého se ukládají rámce historie. Když je buffer plný, stará se o přesun dat do databáze. Rozhraní, které odděluje konkrétní implementaci komunikace s databází, tvoří třída HistoryDbAdapter.

Během serializace se provádí komprese bitmapových snímků a práce s databází také patří mezi časově náročné operace, proto se přesun dat z paměti do databáze provádí v pracovním vlákně pomocí třídy AsyncStoreTask.

AsyncStoreTask

AsyncStoreTask slouží HistoryManageru pro kompresi a přesun dat z bufferu do databáze na pozadí. Tato třída je potomkem pomocné třídy frameworku AsyncTask. Ta slouží k usnadnění provádění výpočetně náročných operací v pracovním vlákně a publikaci výsledků v UI vlákně. Implementuje optimalizovanou správu a pooling pracovních vláken. Z toho by měla kreslící aplikace těžit, neboť bude vytvářet pracovní vlákna pro zápis do databáze poměrně často.

AsyncTask zajišťuje automaticky synchronizaci svých polí. Navíc aby bylo zajištěno, že se aplikace nebude pokoušet číst z databáze ještě neuložená data, jsou metody pracující s databází synchronizovány Java operátorem `synchronized(Object)`.

5.5.2 HistoryDbAdapter

HistoryDbAdapter tvoří oddělené rozhraní pro komunikaci s databází. Podle dostupnosti externího paměťové úložiště (SD karty) určuje cestu pro soubor s SQLite databází. Pokud je externí paměť k dispozici, vytvoří se přímo v pracovním adresáři kreslícího programu. Následně HistoryDbAdapter otevře SQLite připojení a umožní zápis dat.

Pokud není připojena externí paměť použije se HistoryDatabaseHelper, což je třída která dědí z SQLiteOpenHelper. Jedná se o pomocníka pro práci s SQLite databázemi, který automaticky vytváří databázové soubory ve vyhrazeném prostoru interní paměti telefonu. Systém se stará o správu těchto souborů, takže jsou například automaticky odinstalovány spolu s aplikací.

Dále HistoryDbAdapter obsahuje definice jména databáze, tabulky historie s názvy sloupců, SQL dotazů pro potřebné akce nad databází a dalších konstant.

5.5.3 HistoryRenderer

Slouží jako platforma pro renderování informací z historie. Jak bylo uvedeno výše, vykreslování probíhá voláním metody `render (BrushBase, float, float)` jednotlivých segmentů, přičemž informace o barvě, tipu a velikosti hrotu štětce je obsažena v instanci štětce a jeho Stamper objektu.

Renderování historie řídí HistoryManager. HistoryRenderer však obsahuje metody ke zpracování jednotlivých rámců. Pokud mu manažer pošle klíčový snímek, nastaví ho jako plátno. Pokud mu pošle křivku, vykreslí ji s aktuálním nastavením. Pokud pošle rámec s nastavením, renderer dané nastavení na sebe aplikuje a používá ho při dalším vykreslování.

Zároveň sleduje a dokáže najít nejbližší starší rámce s nastavením od zadaného bodu v historii.

6 Návrh a implementace grafického uživatelského rozhraní

Množstvím funkcí lze dostupným konkurenčním aplikacím na platformě Android jen těžko konkurovat. Vypadá to však, že většina z nich se vydala cestou jednoduchého a nepříliš promyšleného uživatelského rozhraní, na které se pak postupně během vývoje nabalovaly další a další funkce.

Tato práce se rozhodla zkusit jít jinou cestou. Zadáním bylo pokusit se zpracovat uživatelské rozhraní více intuitivně a uživatelsky zajímavě. Velký důraz jsem věnoval návrhu, kterému by se pak implementace snažila podle možností platformy přiblížit. Představu mi o něm pomáhal formovat kamarád z Fakulty výtvarných umění VUT, který je zároveň i pokročilým uživatelem několika profesionálních grafickým programů na PC.

Návrh probíhal formou brainstormingu s tužkou a kusem papíru. Vhodné myšlenky jsem pak zapracovával do komplexní představy o vzhledu a funkčnosti uživatelského rozhraní. Tuto představu jsem zpracovával už s dobrým poučením o možnostech platformy.

6.1 Specifikace cílů a návrh vlastností uživatelského rozhraní

Při návrhu uživatelského rozhraní bylo hlavním cílem umožnit uživatelům upravovat prostředí podle jejich stylu práce a specifických potřeb tak, aby měli co největší pohodlí a k nejčastěji používaným funkcím se dostali okamžitě bez procházení různých menu.

Když jsem testoval s výše uvedeným kamarádem dostupné konkurenční aplikace, nejvíce mu vadilo, že neměl žádnou možnost vrátit se k barvě a stylu nástroje, které používal dříve, rozhodl-li se například něco mírně upravit. Původní barvu je sice ve většině aplikací možné získat nástrojem kapátko, nicméně občas je těžké se dobře trefit a pokud kreslíme s průhledností, může být původní odstín pozměněn.

Uživatelské rozhraní by tedy ideálně mělo umožňovat umístit si na snadno dostupné místo tlačítka nejčastěji používaných funkcí a také nejčastěji používané nastavení nástrojů a barev.

Dalším požadavkem byla interaktivita. Každá provedená akce by měla dávat uživateli okamžitou zpětnou vazbu, aby jednak intuitivně poznal co se právě s aplikací děje a jednak aby ho práce s ní bavila.

Jedná se především o různé animace menu, náhledy na prováděné nastavení nástrojů, změny drag & drop stínu podle akce, která se provede puštěním na daný prvek, a podobně.

6.1.1 Hlavní menu

Při návrhu menu bylo nutné vzít v úvahu především budoucí snadnou rozšiřitelnost. Předpokládám, že v programu budou časem přibývat další funkce a mělo by být možné je začlenit do stávajícího uživatelského rozhraní, aniž by nějak rozbíjeli jeho aktuální koncept.

Chceme-li mít menu s velkým množstvím tlačítek na malém displeji, napadají mě v podstatě jen dvě možnosti jak toho dosáhnout. První je hierarchické členění do podmenu a druhá spočívá v posouvání a listování obsahem. Optimální je podle mě obě možnosti spojit a vytvořit systematické dělení do skupin funkcí a zároveň umožnit listování a tím pádem neomezený počet prvků v menu.

Android framework nemá žádný widget, který by přímo vyhovoval mé představě. Obsahuje však nástroje které značně ulehčí implementaci takové komponenty. Výsledek by měl vypadat jako na obrázku 21. Položeným prstem lze posouvat do stran. Při *„fling“* gestu (rychlý tah prstem) má navíc pohyb setrvačnost.

Obrázek 21: Widget posuvné menu

6.1.2 Konfigurovatelná pracovní plocha

Uživatel má mít možnost umístit si na libovolné místo uživatelského rozhraní zkratky na nejčastěji používané funkce.

Funkce programu se vyvolává převážně tlačítkem, proto by bylo dobré vytvořit mřížku, do které se budou umísťovat buď přímo tlačítka, nebo nějaký typ obalového widgetu. V rámci mřížky a dalších míst programu, jako je posuvné menu, by mělo jít prvky libovolně přemisťovat. Je ale potřeba dbát na to, aby nevznikali v menu duplicitní tlačítka a na druhé straně, aby nebylo možné se tlačítka nějaké funkce zbavit natrvalo.

Mřížka usnadní také detekci kolizí prvků. Pokud by se umístění uvádělo absolutně pomocí souřadnic, bylo by nutné dělat detekci kolizí v rámci pixelů, což by bylo zbytečně náročné. U mřížky stačí udržovat binární informaci o obsazenosti buněk a při umísťování prvku ji projít jednoduchým cyklem, který zjistí, zda jsou všechny buňky volné.

Plocha by měla mít koš na prvky, které už na ní nechceme. Zároveň by bylo dobré, aby byl zobrazen pouze v módu přemisťování a nezabíral zbytečně místo na ploše.

6.1.3 Volby nástrojů a barev

Od začátku návrhu bylo na první pohled jasné, že volba barvy se nejlépe zpracuje formou specializovaného dialogu. Na druhou stranu u voleb velikosti a průhlednosti nástroje jsem chtěl uživatelům umožnit umístit si je také libovolně na plochu. Silným argumentem proti tomuto řešení je ale nedostatek místa u mnoha zařízení. Ovládací táhla by pak musela být velmi malá, aby nezabírala většinu pracovní plochy, a tím by utrpěl komfort ovládání.

Lepší řešení je tedy podle mě umístit všechna nastavení týkající se nástrojů do jednoho specializovaného dialogu a volby barev do druhého dialogu.

V dialogu si uživatel připraví často používaná nastavení a ta pak bude mít ve formě tlačítek dostupná na hlavní ploše. Zároveň může být v dialogu přítomna rozsáhlejší knihovna uživatelských nastavení nebo automaticky generovaná historie umožňující návrat k použitým volbám, které nebyly zapamatovány.

Dialog nástrojů musí obsahovat náhled, který okamžitě ukazuje, jak prováděné nastavení ovlivní bitmapu štětce. Interaktivní náhled, který umožní dotykem ovládání alespoň části parametrů, jistě také přidá na dobrém uživatelském zážitku. Užitečný by mohl být i náhled na výslednou stopu nástroje. Bitmapa hrotu je asi důležitější a měla by být na prvním místě, přičemž by mělo být umožněno například přepnutí na náhled stopy.

Pro volbu barev je dle mého názoru nejlepší barevné kolo podle modelu HSV (nebo také někdy HSB). Dle teorie barevných modelů [8] je pro lidské vnímání více intuitivní než standardně používaná RGB reprezentace.

V barevném kole jsou po obvodu rozmístěny hodnoty barevného tónu (*hue*). Uvnitř kruhu je pak trojúhelník nebo čtverec. V něm jsou rozmístěny kombinace hodnot sytosti (*saturation*) a jasu (*brightness* nebo *value*).

Tento princip využívá i většina grafických aplikací jak na PC, tak na mobilních zařízeních.

6.2 Implementace uživatelského rozhraní

K dosažení výše specifikovaných cílů jsou prvky uživatelského rozhraní připravené v knihovnách platformy nedostatečné. Bylo nutné implementovat několik vlastních widgetů a rozvržení prvků, které by podporovaly požadované chování. Jako výchozí bod slouží třídy třídy View a ViewGroup. Pokud nechceme začínat úplně od začátku, můžeme rozšířit i jejich podtřídy, které už implementují část požadovaného chování.

6.2.1 Obecné posuvné menu a hlavní menu

Na obrázku 22 vidíme diagram tříd souvisejících s implementací posuvných menu. Jako výchozí bod slouží abstraktní třída ViewGroup. Menu bude totiž složeno z tlačítek, které budou vycházet z některé Android implementace (Button nebo ImageButton). Jako tlačítko může sloužit jednoduše jakákoli podtřída View, ale nemá pak funkce jako načítání obrázku z XML souborů zdrojů.

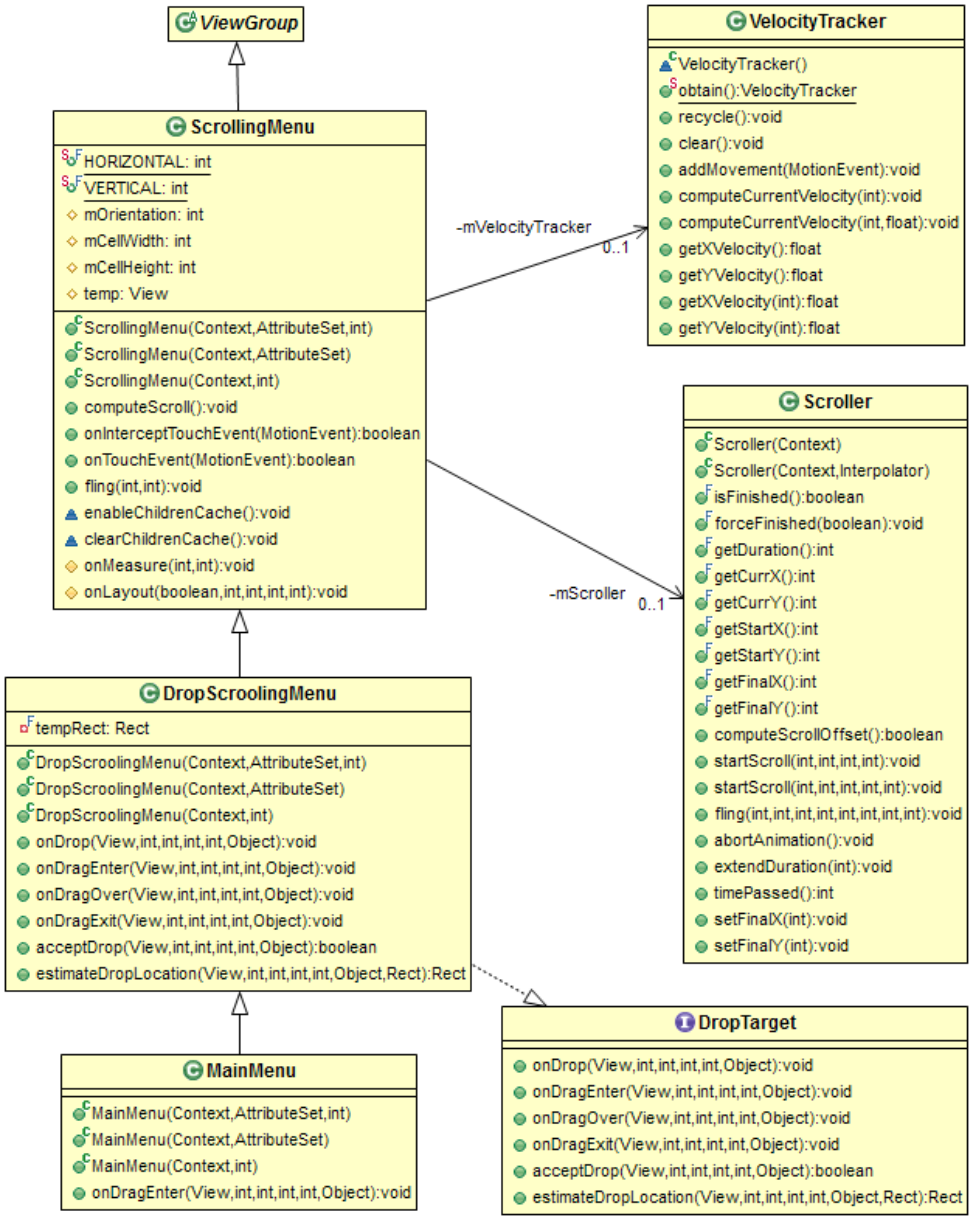
Vlastní chování menu, včetně umísťování potomků, je implementováno v třídě ScrollingMenu. Můžeme mít dvě rozložení. Vertikální umístí potomky do sloupce nad sebe a horizontální do řádku vedle sebe. Prvky rozmístí tak, aby se jich vešlo do přiděleného prostoru co nejvíce. Zbylou mezeru rovnoměrně rozdělí.

Rolování obsahu widgetu se provádí metodou `View.scrollTo(int, int)`. Obsah widgetu je posunut o zadaný offset.

Pokud posouváme obsah widgetu s prstem na obrazovce, offsety se spočítají jako rozdíl souřadnic pohybových událostí. Pro implementaci setrvačnosti po rychlém (*fling*) gestu je už nutný složitější postup.

K výpočtu hodnot offsetu při animování rolování se setrvačností se používá Scroller objekt. Animace se nejlépe implementuje pomocí metody `computeScroll()`, která je volána rodičem při překreslování widgetu. Pokud do implementace metody umístíme zneplatnění objektu metodou `postInvalidate()`, vytvoří se smyčka, která bude animovat posun. Ukončovací podmínkou smyčky bude offset pro posunutí menší nebo roven nule.

Scroller objekt potřebuje k zahájení animace počáteční rychlost pohybu. Tu můžeme získat bez velké námahy z objektu třídy VelocityTracker, který jsme předtím nakrmili daty z pohybových událostí.



Obrázek 22: Diagram tříd menu widgetů

V konstruktoru Scroller objektu je možné definovat interpolátor, což je objekt, který udává časový průběh křivky, podle které se interpoluje průběh rolování. Já jsem chtěl dosáhnout velkého utlumení pohybu, aby byla práce s menu přesnější. Při použití výchozího interpolátoru se totiž i poměrně pomalým gestem menu posunulo o velkou vzdálenost.

Zvolil jsem DecelerateInterpolator, jehož průběh vypadá jako otočená $y=x^2$ parabola. Na počátku je tempo změny rychlosti velké a postupně zvolňuje. Aplikací faktoru vyššího než jedna můžeme posílit efekt zvolnění (začíná ještě rychleji a končí ještě pomaleji).

Scroller objektu říkáme, za jak dlouhou dobu má provést přesun o danou vzdálenost a interpolátor pak udává časovou funkci. Při ladění setrvačnosti jsem chtěl mít konstantní dobu interpolace, ale zároveň i výchozí rychlost stejnou jako je rychlost gesta. Je tedy nutné zadávat Scroller objektu správnou hodnotu vzdálenosti tak, aby výchozí rychlost odpovídala rychlosti gesta. Spočítá se jako 6.1.

$$\Delta l = \frac{v * t}{factor} \tag{6.1}$$

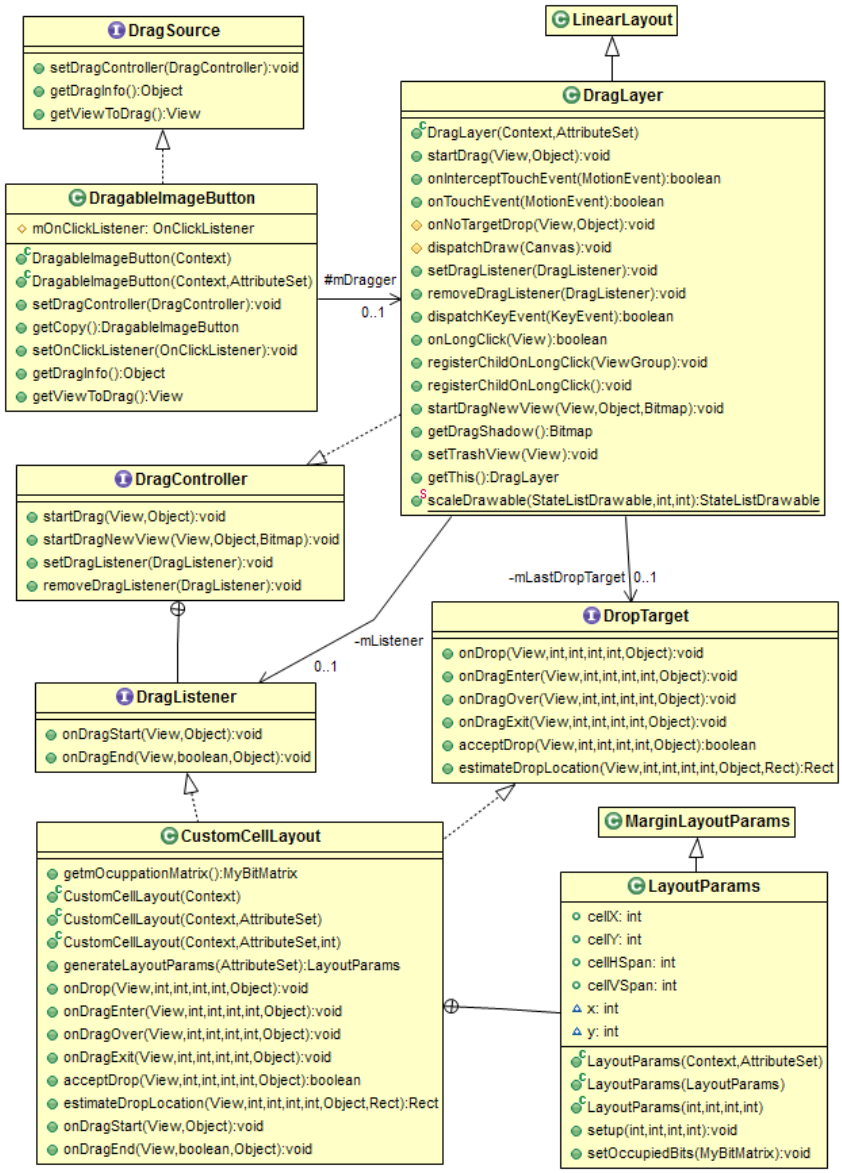
DropScrollingMenu rozšiřuje možnosti o implementaci rozhraní DropTarget. Menu pak může přijímat nové prvky, které na něj přemístíme.

MainMenu je finální implementace hlavního menu, která už má pevně zakódované chování při interakci s dalšími widgety uživatelského rozhraní.

6.2.2 Drag & Drop

Platforma Android poskytuje podporu pro drag & drop funkce od API verze 11. Já jsem se však už v úvodu vývoje rozhodl cílit na verzi 8 a vyšší. Musel jsem si tedy podporu implementovat sám.

Výborným zdrojem pro začátek je zdrojový kód výchozího Launcheru [9]. Ke stažení je nutný git.



Obrázek 23: Diagram tříd Drag & Drop a pracovní plochy

Na obrázku 23 vidíme diagram tříd celého systému. Obecně se skládá ze zdrojového rozhraní, které vyvolá drag & drop proces, dále cílového rozhraní, které dokáže zpracovat přesouvaná data a řídící jednotky.

V našem případě je přesouvaná informace widget uživatelského rozhraní. Přesouvá se zpravidla mezi různými nadřazenými kontejnery.

Rozhraní řídící jednotky implementuje třída DragLayer. Ta se dále stará o zpracování pohybových událostí a vykreslování bitmapy, která představuje přemísťovaný objekt (*drag shadow*). DragLayer je nepřímým potomkem ViewGroup a je nutné, aby se bitmapa zobrazovala přes veškerý ostatní obsah, proto se vykreslení provádí v metodě `dispatchDraw(Canvas)` až poté, kdy je zavolána implementace této metody v rodiči.

6.2.3 Plocha pro umístění widgetů

Třída CustomCellLayout se stará o rozložení potomků podle mřížky. V XML je možné stanovit velikost stran buňky. Zvolil jsem hodnotu 48dp což je podle [10] minimální velikost, do které se lze pohodlně trefovat prstem.

Překrývání widgetů na ploše zabraňuje jednoduchá detekce kolizí. Každá buňka má přiřazenu binární informaci o obsazenosti a při uvolnění tahu (*drop*) se kontroluje, jestli jsou buňky v požadované oblasti volné. Informace uchovává objekt třídy MyBitMatrix, která obaluje třídu BitSet a přidává metody pro snadnou interpretaci pole bitů maticí bitů.

Při umísťování widgetů se příslušné metody řídí podle objektů třídy LayoutParams. Každý potomek View může mít pro tyto účely přiřazeného potomka View.LayoutParams. Objekt obsahuje pozici levé horní buňky, šířku a výšku. Vše se uvádí v souřadnicích mřížky. Přepočet na pixely provádí CustomCellLayout podle hodnot XML atributů.

6.2.4 Dialog nástrojů

Rozložení dialogu nástrojů je definováno v souboru brush_dialog.xml. Skládá se z posuvného menu s tlačítky pro volbu typu nástroje, náhledu bitmapy štětce, boxu pro umístění vytvořené kombinace parametrů a několika posuvníků pro změnu nastavení.

To vše je umístěno v DragLayer kontejneru, díky čemuž je možné pomocí drag & drop přesouvat nastavení z náhledu do boxu.

Náhled na bitmapu štětce je implementován v třídě BrushTipPreview. Pohyb dvěma prsty k sobě a od sebe v X-ové ose umožňuje nastavení velikosti a v Y-ové nastavení průhlednosti.

Kontejner pro ukládání nastavení představuje třída BrushDialogSettingsBox. Je potomkem třídy BoxLayout, která se chová velmi podobně jako CustomCellLayout, jen je zjednodušená, neboť neumožňuje větší šířku widgetů než jedno pole.

BrushDialogSettingsBox je provázán s tlačítkem pro spuštění dialogu nástrojů. Při umístění na pracovní plochu se tlačítko přemění na box obsahující definovaná nastavení.

6.3 Výsledný vzhled a zhodnocení

Na připravené komponenty byl posléze aplikován grafický styl. Za pomoc s jeho návrhem a výrobou bych rád poděkoval Štěpánu Křížkovi z FaVU. Při jeho návrhu jsme chtěli docílit především jednoduchosti. Dalším požadavkem byl jednotný vizuální styl celé aplikace.

Rozhodli jsme se hodně pracovat s průsvitností a využít také v poslední době velmi moderní zaoblené okraje. Tlačítka jsou označena jednoduchými černobílými symboly. Celkově je vizuální styl založen na kontrastu černé a bílé s využitím neutrální šedé především pro pozadí.

6.3.1 Hlavní menu a pracovní plocha

Výsledný vzhled po aplikaci grafického stylu je na obrázku 24. Nutno podotknout, že obrazovka byla sejmuta ze zařízení Dell Streak, které má fyzicky velmi velký displej.

Obrázek 24: Výsledný vzhled hlavního menu a pracovní plochy

V hlavním menu na vrchu obrazovky vidíme tlačítka pro (zleva) dialog nástrojů, volbu barvy, historii, otevření souboru, uložení práce, vycentrování plátna na velikost obrazovky a vymazání obsahu.

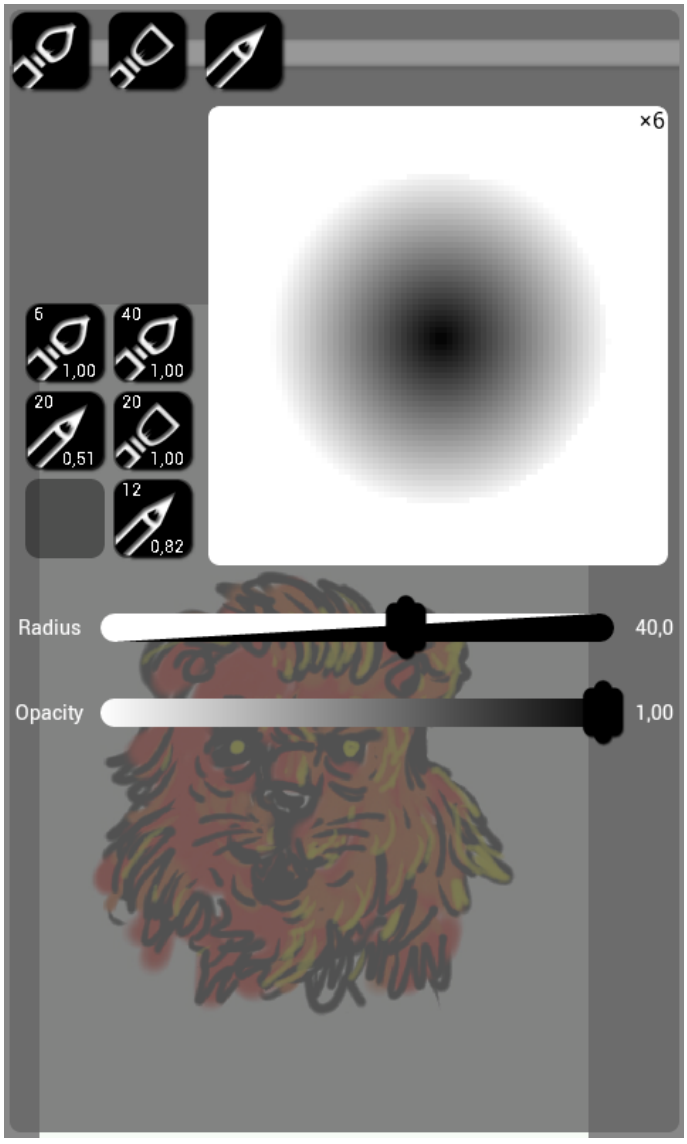
Na pracovní ploše je umístěn box s připravenými nastaveními nástrojů a často používaná tlačítka zpět a vycentrování plátna.

Myslím, že většinu cílů stanovených při návrhu se podařilo zpracovat velmi kvalitně. Menu se chová velmi příjemně a díky posouvání obsahu do stran by neměl být problém ani s optimalizací na menší zařízení nebo s přidáváním nových funkcí.

Tlačítka je v menu možné libovolně přerovnat a přizpůsobit si tak jejich pořadí. Pokud tlačítko vytáhneme na plochu, vytvoří si zástupce. V případě tlačítka nástrojů se při přesunu na pracovní plochu přemění na box s nastaveními.

6.3.2 Dialog nástrojů

Výsledný vzhled dialogu zachycuje obrázek 25. Podařilo se dosáhnout přesně takového vzhledu, jaký byl předem navržen v Ilustratoru. Chování náhledu při gestech není úplně optimální. Osy trochu přeskakují. Zřejmě by bylo potřeba více vyladit práh citlivosti.



Obrázek 25: Dialog nástrojů

7 Závěr

V rámci této práce jsem připravil základní funkce kreslícího programu a implementoval většinu nejdůležitějších částí uživatelského rozhraní. Bohužel se před dokončením tohoto textu nepodařilo aplikaci zveřejnit a získat větší zpětnou vazbu od uživatelů. Nechtěl jsem totiž ohrozit dobré hodnocení aplikace tím, že bych ji vypustil před dokončením všech nutných funkcí a odladěním chyb.

Vlastnosti aplikace však byly během vývoje průběžně testovány pokročilým uživatelem grafických programů a jeho připomínky byly zapracovány.

Jako hlavní úspěch hodnotím kvalitně zpracovaný interpolátor. Sám o sobě sice nestačí, aby aplikace předstihla konkurenci, pomůže však zvýšit vizuální kvalitu jakéhokoli nástroje. Dobře zpracovaný zoom je nutnost a jsem rád že funguje minimálně stejně dobře, jako u předních kreslících programů na platformě.

Historie je v této podobě funkční, má však ještě své problémy. Bude potřeba dále optimalizovat rychlost vykreslování a provést zátěžové testování. Momentálně také není nijak vyřešen problém nedostatku paměti pro historická data. Ta však mohou při rozsáhlejší práci snadno nadměrně narůstat.

Vzhled uživatelského rozhraní je myslím příjemný. Bylo dosaženo také velmi dobré konfigurovatelnosti, což byl jeden z hlavních požadavků. Ovládání je dostatečně jednoduché na to, aby ho dokázal okamžitě uchopit i člověk bez předchozí zkušenosti s podobným programem. Problém je především s drag & drop, kdy uživatelé nevědí, že je k dispozici.

Ze základních funkcí, které bude nutné před zveřejněním doplnit, bych zmínil například nástroje kapátko, guma nebo plechovka barvy. Dále ve stávajícím řešení chybí dobře zpracovaný dialog barev, který by byl konzistentní s dialogem nástrojů. Prozatím je použito provizorní řešení z volně dostupné knihovny. Rozpracovanou novou verzi lze však nalézt ve zdrojových kódech.

Po zveřejnění plánuji na projektu dále pracovat a rozšiřovat dostupné funkce. Zcela jistě plánuji dodělat funkci přehrávání průběhu kreslení. Architektura historie byla navržena s ohledem na snadnou integraci této funkce, takže by implementace v této fázi už měla být relativně snadná. Další možnosti rozšíření vidím například v práci s vrstvami a v neposlední řadě v neustálém vylepšování uživatelského rozhraní.

Literatura

[1] Křivky a plochy. ŽÁRA Jiří, BENEŠ Bedřich, SOCHOR Jiří a FELKEL Petr. *Moderní počítačová grafika*. 2. přep. a roz. vydání. Brno: Computer Press, 2004, s. 177-198. ISBN 80-251-0454-0.

[2] BRANSON, Kristin. *A Practical Review of Uniform B-Splines* [online]. San Diego, 3.11.2004 [cit. 4.1.2012]. Dostupné z WWW: http://vision.ucsd.edu/~kbranson/research/bsplines.html. UCSD Jacobs School of engineering

[3] BAKER, Kirby. *Cubic spline curves* [online]. Los Angeles, 2002 [cit. 6.1.2012]. Dostupné z WWW: http://www.math.ucla.edu/~baker/149.1.02w/handouts/dd_splines.pdf. UCLA University of California

[4] MAŁCZAK, Mateusz. *Quadratic Bezier curve length*. *segfault labs* [online]. [cit. 7.1.2012]. Dostupné z WWW: http://segfaultlabs.com/docs/quadratic-bezier-curve-length

[5] TIŠNOVSKÝ, Pavel. *Interaktivní editor afinních transformací*. Brno, 1999. Dostupné z WWW: http://www.fit.vutbr.cz/~tisnovpa/publikace/diplomka/doc/node1.html. Diplomová práce. Vysoké učení technické v Brně

[6] VOGEL, Lars. *Android Development Tutorial*. *Vogella* [online]. 04.07.2009, 17.04.2012. [cit. 24.4.2012]. Dostupné z WWW: http://www.vogella.com/articles/Android/article.html

[7] *Android developers*. [online]. [cit. 2.5.2012]. Dostupné z WWW: http://developer.android.com

[8] Achromatic and colored light. D. FOLEY James, VAN DAM Andries, K. FEINER Stephen, F. HUGHES John. *Computer Graphics: Principles and practise*. 2nd edition. Boston: Addison-Wesley, 2003, s. 563-603. ISBN 0-201-84840-6.

[9] The Android Open Source Project. *Zdrojové kódy výchozího Launcheru*. [online]. 2008, [cit. 12.5.2012]. Dostupné z WWW: http://android.git.kernel.org/?p=platform/packages/apps/Launcher2.git;a=summary

[10] *Android Design*. [online]. [cit. 12.5.2012]. Dostupné z WWW: http://developer.android.com/design/style/metrics-grids.html

Seznam příloh

Příloha 1. Implementace algoritmu tahu štětcem

```
/**
* Render line
* @param begin First line point.
* @param end Second line point
* @param stampingBrush Class implementing stamper interface.
* @param leftOver Part from previous segment not covered by stamping
* @param spacing Distance between two stamps
* @param position number between 0-1 determining stamp position in path segment,
      -1 to indicate segment is line and position must be determined here
* @param scale Scale in ScreeToCanvas matrix in matrix logic
* @return leftover for next segment.
*/
public static float render (PointF begin,PointF end,BrushBase
      stampingBrush,float leftOver,float position,float scale){
      //determine slope
      float deltaX = end.x - begin.x;
      float deltaY = end.y - begin.y;
      //normalize delta vector
      float distance = FloatMath.sqrt(deltaX*deltaX + deltaY*deltaY);
      float stepX = 0;
      float stepY = 0;
      if(distance > 0){
            float invertDist = 1/distance;
            stepX = deltaX * invertDist;
            stepY = deltaY * invertDist;
      }
      float offsetX = 0;
      float offsetY = 0;
      float totalDistance = leftOver + distance;

      float spacing = stampingBrush.getSpacing()*scale;
      //****position computing
      float pos = 0f;
      float step = spacing/totalDistance;
      float lostep = -1f;

      while(totalDistance >= spacing){
            //increment offset and stamp
            if(leftOver > 0){
                  offsetX += stepX * (spacing - leftOver);
                  offsetY += stepY * (spacing - leftOver);

                  //position
                  lostep = (spacing - leftOver)/totalDistance;

                  leftOver -= spacing;
            }
            else{
                  offsetX += stepX*spacing;
                  offsetY += stepY*spacing;
            }

            if(position >= 0){ //position is given from another segment which is
                              using this function to render part of himself
                  pos = position;
            }
```

```
            else{ //position is -1, rendered segment is line and position must
                  be computed here
                  if(lostep > 0){
                        pos = lostep;
                        lostep = -1f;
                  }
                  else{
                        pos += step;
                  }
            }

            stampingBrush.stampAt(new PointF(begin.x + offsetX,begin.y +
            offsetY),pos);

            totalDistance -= spacing;
      }
      return totalDistance;
}
```

Příloha 2. Obsah CD

- Zdrojové texty
- Spustitelný binární .apk soubor
- Javadoc dokumentace
- Demonstrační video
- Tato práce ve formátu .odt
- Diagram všech tříd